```
alert_messages: data_in

BODY:
If the alert_queue is empty, the Add Smoke Detector Alert
to Queue function shall retrieve the associated alert_message,
based on detection_type, from the alert_messages store and
send it to the display. Otherwise, it will add the sensor_id
to the alert_queue for subsequent display.

If the detection_type = “SMOKE”, this function shall set the
detector_lamp_command to “RED”. If the detection_type = “NO SMOKE”
it shall set the detector_lamp_command to “GREEN”. This function
shall send the detector_lamp_command to the lamp corresponding to
the sensor_id.


NAME:
4.4

TITLE:
Add Fuel Alert to Queue

INPUT/OUTPUT:
alert_queue: data_out
alert_messages: data_in
fuel_lamp_command: control_out

BODY:
If the alert_queue is empty, the Add Fuel Alert to
Queue function shall retrieve the associated alert_message
from the alert_messages store and send it to the display.
Otherwise, it will add the sensor_id to the alert_queue for
subsequent display. It shall also set fuel_lamp_command to
“RED” and send it to the fuel lamp.
```

Figure 3.27 P-specs for primitive functions of Generate Alarm

```
NAME:
4.1
TITLE:
Add Out-of-Range Alert to Queue
INPUT/OUTPUT:
sensor_id: data_in
alert_queue: data_out
alert_messages: data_in
display_data.alert_message: data_out
sensor_lamp_command: control_out
BODY:
If the alert_queue is empty, the Add Out-of-Range Alert
to Queue function shall retrieve the associated alert_message
from the alert_messages store and send it to the display.
Otherwise, it will add the sensor_id to the alert_queue for
subsequent display. It shall also set sensor_lamp_command to
“RED” and send it to the lamp corresponding to the sensor_id.
NAME:
4.2
TITLE:
Reset Lamp
INPUT/OUTPUT:
sensor_id: data_in
detector_lamp_command: control_out

fuel_lamp_command: control_out
sensor_lamp_command: control_out
BODY:
The Reset Alarm function shall set either the sensor_lamp_command,
the detector_lamp_command, or the fuel_lamp_command to
“GREEN” based on the sensor_type and send it to the lamp
corresponding to the sensor_id.

Note that this operation is independent of when the pilot
acknowledges the corresponding alert message that is on display.

NAME:
4.3
TITLE:
Add Smoke Detector Alert to Queue
INPUT/OUTPUT:
detection_type: data_in
alert_queue: data_out
sensor_id: data_in
detector_lamp_command: control_out
```

Figure 3.29 4-S1 Generate Alarm STD

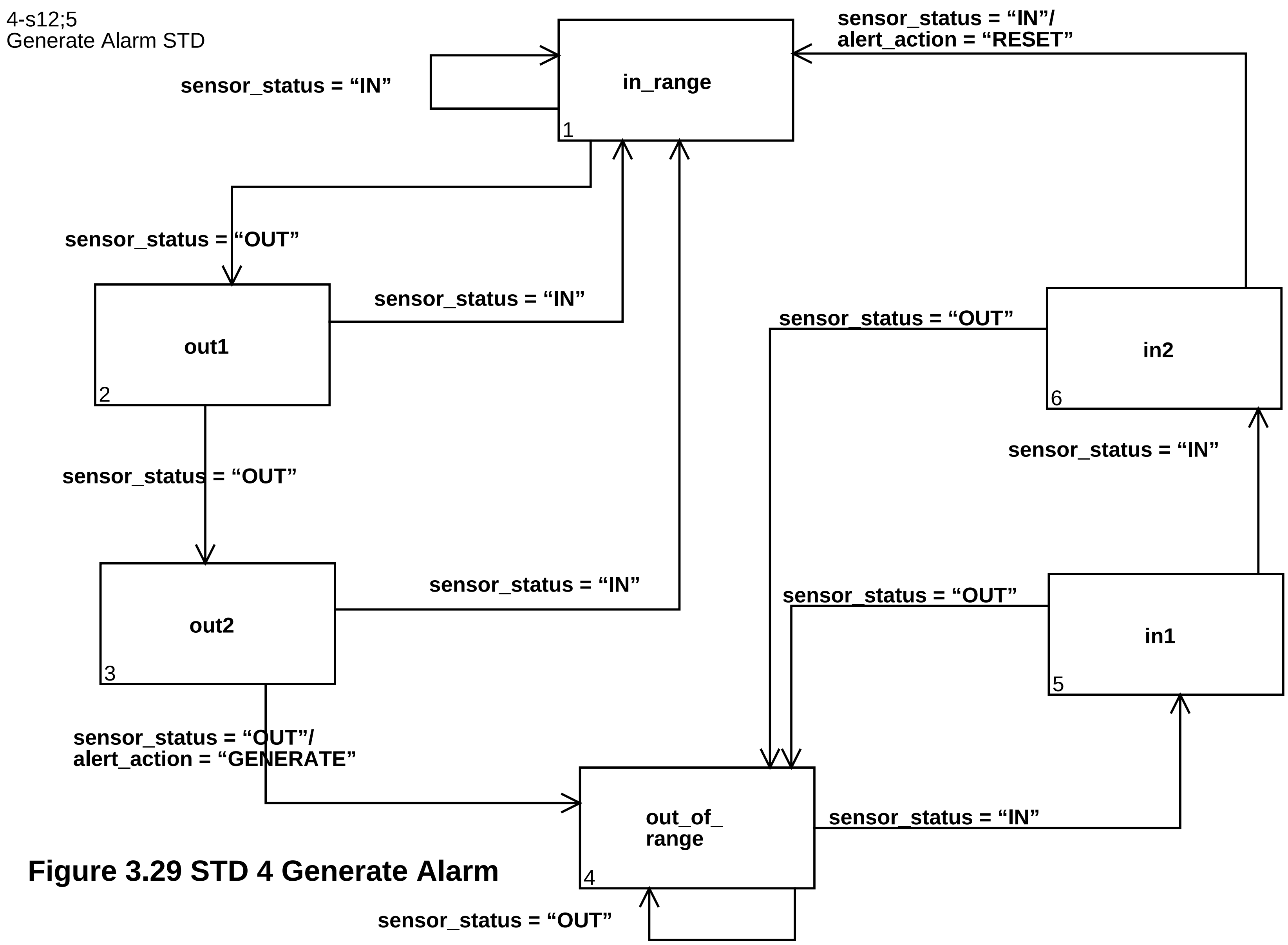


Figure 3.29 STD 4 Generate Alarm

Figure 3.28 4-S2 Generate Alarm PAT

4-s2;8
Generate Alarm PAT

Notes: 1. smoke_detection is independent of out-of-range condition.
2. fuel capacity can be either out-of-range or low.

alert_ action	smoke_ detection	fuel_status	1	2	3	4
			Add Out-of-Range Alert to Queue	*Reset Lamp*	*Add Detector Alert to Queue*	*Add Fuel Alert to Queue*
"GENERATE"			1	0		
"RESET"			0	1	0	0
	"TRUE"			0	1	
	"FALSE"			1	0	
		"LOW"		0		1

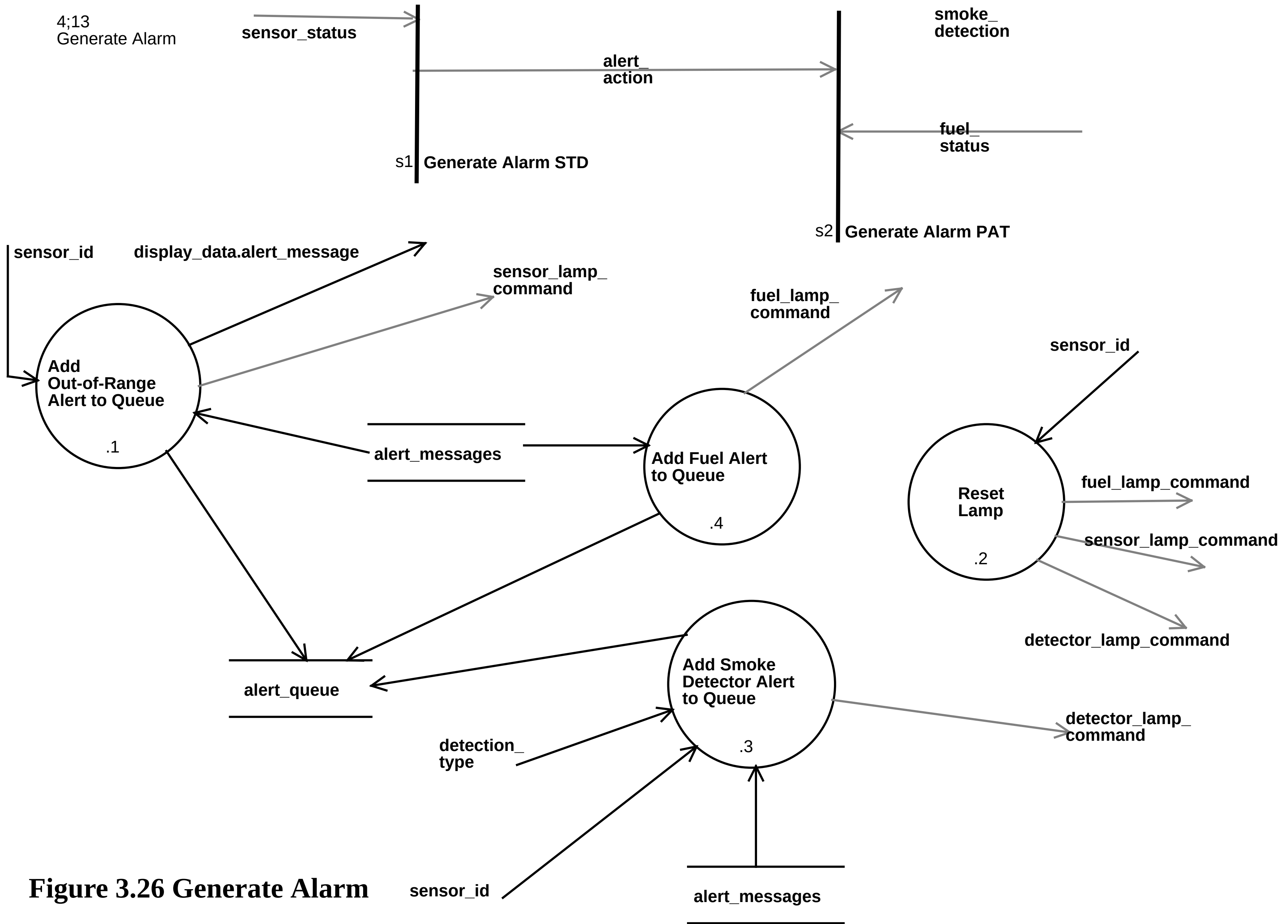


Figure 3.26 Generate Alarm

```
fuel_data_received: control_out
sensor_reading: data_out
time_of_day: data_in
      BODY:
      Upon receipt of the sensor_data, this function shall update
      the sensor_reading store with the sensor_data and the
      time_of_day received, and set sensor_data_received to “TRUE”.
      If the sensor_type = “F”, this function shall also set
      fuel_data_received to “TRUE”.
```

```
range_constants: data_in
time_of_day: data_in

BODY:
For the sensor being polled, the Record Time-out function
shall update the sensor_reading store as follows:

1. Set time_received to time_of_day.

2. Based on the range_constants, set the sensor_value
    to an out-of-range value. (Note: a Time-out condition
    shall be treated as an out-of-range condition.)

NAME:
1.2

TITLE:
Determine Range

INPUT/OUTPUT:
sensor_reading: data_in
range_constants: data_in
sensor_status: control_out
sensor_id: data_out

BODY:
The Determine Range function shall compare each
sensor's sensor_value with the corresponding
range_constants for the sensor. If the value is
out of range, it shall set sensor_status to “OUT”,
otherwise, it shall set sensor_status to “IN”.
It will also output the sensor_id along with the
status.

NAME:
1.3

TITLE:
Determine Fuel Capacity

INPUT/OUTPUT:
sensor_reading: data_in
fuel_status: control_out

BODY:
The Determine Fuel Capacity function shall
compare the sensor_value for the fuel tank
entry of the sensor_reading store with a TBD maximum
fuel amount. If the value is less than 10% of
maximum, this function shall set fuel_status
to “LOW”.

NAME:
1.4

TITLE:
Receive Sensor Data

INPUT/OUTPUT:
sensor_data: data_in
sensor_data_received: control_out
```

NAME:
1.1
TITLE:
Record Time-out
INPUT/OUTPUT:
sensor_reading: data_out

1-s2;2
Monitor Sensor SEM

States/ Events	sensor_data_ received	Time-out	one_second_ interrupt
Idle	*don't care*	/Idle	reading_request; set_timer/ Polling Sensor
Polling Sensor	/Idle	record_timeout; sensor_data_ received/Idle	*don't care*

1-s1;4
Monitor Sensor PAT

sensor_data_ received	fuel_data_ received	record_ Time-out	2	3	1
“TRUE”			**1**		
	“TRUE”			**1**	
		“TRUE”	**2**		**1**

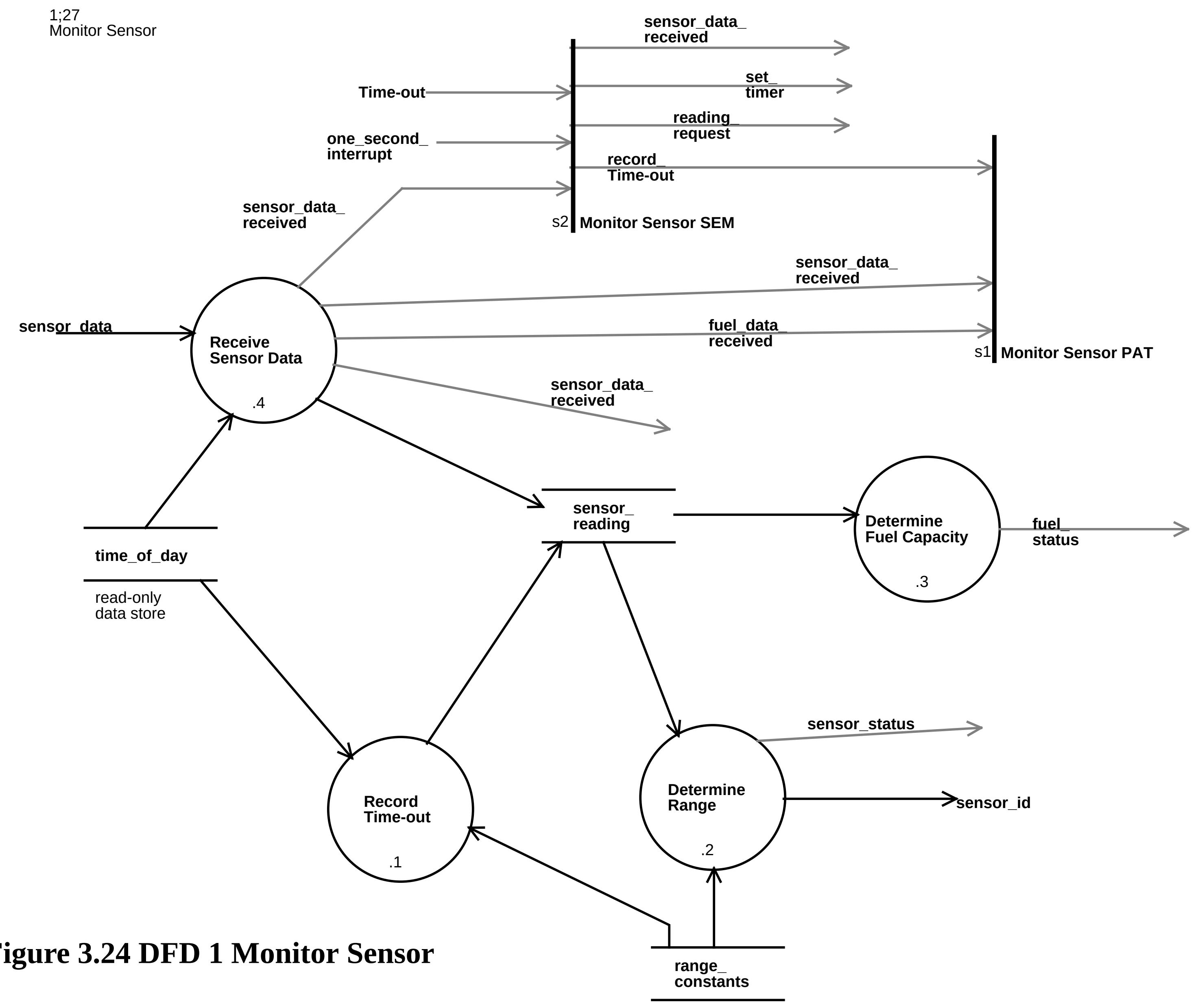
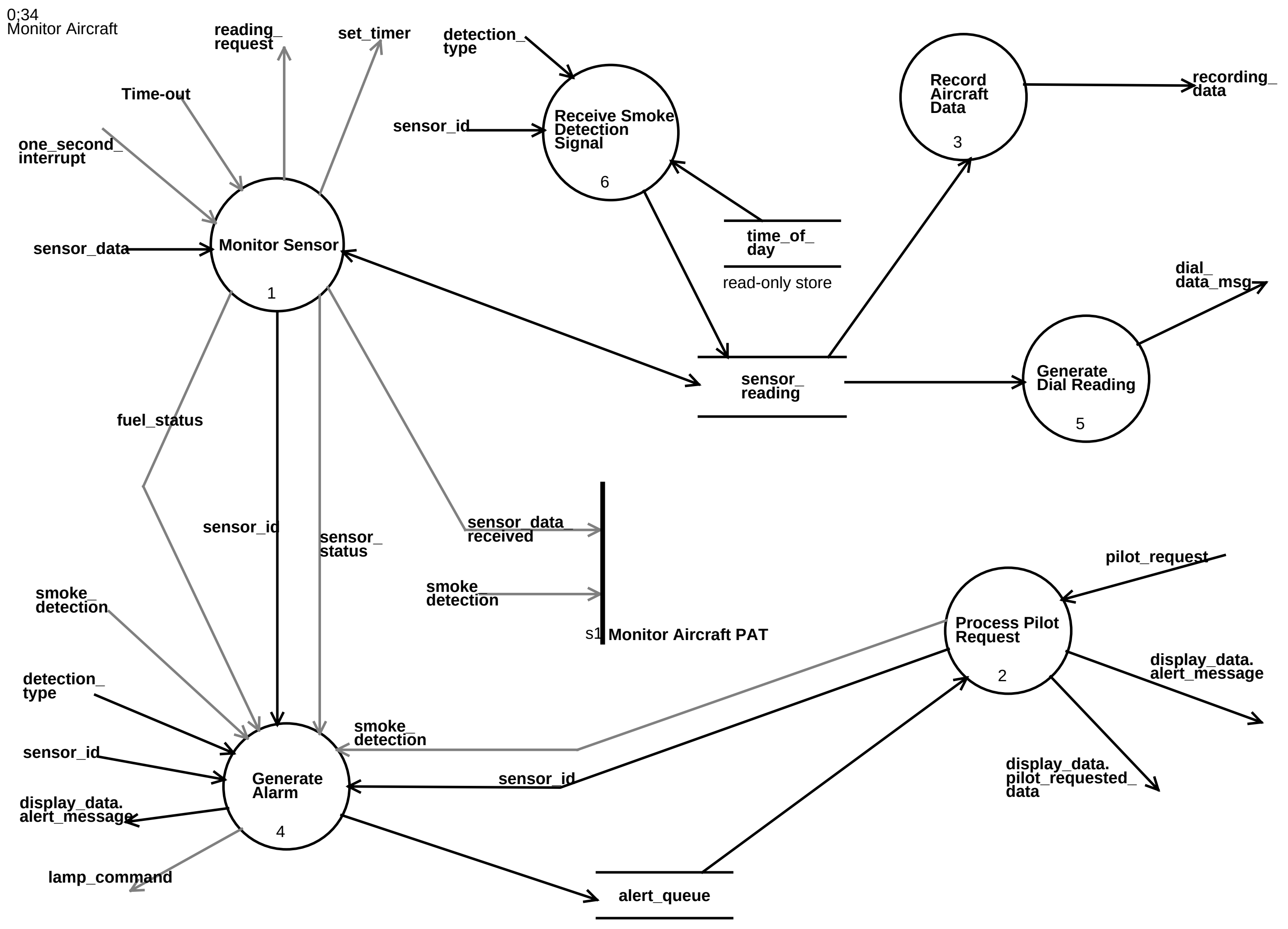


Figure 3.24 DFD 1 Monitor Sensor

Figure 3.23 DFD 0 Monitor Aircraft



back to state Idle occurs with a sensor_data_received event or a Time-out event. The state machine is specified for monitoring only one sensor. Since there are several sensors in this application, each sensor should have its own instance of the controller S2. This is actually why object-oriented analysis was proposed to specify a class of objects where several instances of the class can coexist. Figure 3.25 (c) shows the P-specs for the primitive processes 1.1, 1.2, 1.3, and 1.4, respectively.

The Generate Alarm function of DFD 0 is specified further using DFD 4 as shown in Figure 3.26. The read only alert_messages store contains typical alert messages to be displayed to the pilot when an out of range sensor reading (from the temperature or pressure sensors) is detected, or a fuel alert is to be generated, or a smoke detector alert is signaled. The alert_queue store contained queued alter messages to be displayed in sequence to the pilot by the process_pilot_request function in DFD 0. The same alert_queue store is also shown in DFD 0 in Figure 3.23.

The P-specs for the four functions shown in Figure 3.26 are shown in Figure 3.27. Figures 3.28 and Figure 3.29 show the C-specs for controllers S2 and S1 respectively. The PAT shown in Figure 3.28 activates or deactivates the function in DFD 4 depending on the values for the signals alert_action, smoke detection, and fuel_status. The alert_action signal, produced by controller S2, is asserted to “GENERATE” in the event of receiving three consecutive out of range sensor reading (from the pressure or temperature sensors). The signal is asserted to “RESET” when three consecutive sensor readings are within the range of sensor readings as shown in Figure 3.29.

- upper_limit (data flow, pel) =

  * The upper limit in an allowable range

  for a sensor. *.

Figure 3.23 shows the first level decomposition of AMS presented in DFD 0. Six functions are specified. The first function deals with the process of monitoring the sensors. The sensors pooling sequence, timer setting, the one second interrupt input, and the reading and storing sensor data are all activities performed by the Monitor sensor function. The second function deals with inputting and processing pilot requests. It also sends pilot requested data to the CRT display. This structure is typical in many real-time systems where synchronous or periodical events (such as sensor reading) are handled by one function, and asynchronous events such as pilot requests are handled by a separate function. This facilitates the detailed specification of each of these functions. The third function is a simple function producing a recording_data output from the sensor_reading store. The fourth function, Generate Alarm, deals with the processed sensor information from function Monitor Sensors and produces warning and alarm signals and data to the CRT display, to the lamps, and to an alert_queue store. The fifth function is a simple function producing output data for the dials. The last function is dedicated to reading and storing smoke detection information. It is activated by the controller S1 when a smoke detection signal is asserted. The controller also activates a subset of functions when the sensor_data_received signal is asserted, and yet another subset of processes must be active all the time and hence they are not controlled by S1. The C-spec sheet specifying the PAT for S1 is left as an exercise for the reader.

Figure 3.24 Shows the lower level DFD 1 of the Monitor Sensor function. It is clear that the large number of inputs and outputs of this function would make it complex enough to necessitate the development of a lower level DFD. The same sensor reading and time_of_day stores in DFD 0 is shown again in this DFD. The input function Receive Sensor Data must be active all the time to read and store the input data. It also produces output control signals indicating that data has been received. These signals are sensor_data_received and fuel_data_received. There is some ambiguity here in the names of these two signals. This ambiguity will be come clear when we discuss the C-specs for controllers S1 and S2. The Record Time-out function is activated when the record_timeout signal is asserted by the controller S2. It records in the sensor_reading store a time out event tagged with the current time for the sensor being polled. Function Determine Fuel Capacity is activated by S1 only when the fuel_data_received signal is asserted. Similarly, function Determine range is activated only when the sensor_data_received signal is asserted.

The C-spec sheets for controllers S1 and S2 of DFD1 are shown in Figure 3.25 (a), and Figure 3.25 (b), respectively. The state machine specified by the SEM in Figure 3.25 (b) consists of two states, Idle and Polling Sensor, for the Monitor Sensor process. The one_second_interrupt event triggers a reading_request and a set_timer signals to be asserted and results to a transition from the Idle state to Polling Sensor state. The transition

- temperature_data (data flow, pel) =

  * Temperature returned from the engine temperature

  sensor. *.

- temperature_data_buffer (store, pel) =

  * Temperature returned from the engine temperature

  sensor which has been placed in the buffer. *.

- temperature_data_received (control flow, pel) =

  ["TRUE" | "FALSE"].

  * Indicates whether or not a response was received

  from the polling of the temperature sensor. *

- test_smoke_detector (data flow, del) =

  ["TRUE" | "FALSE"].

  * Signal indicating that the pilot wants to test

  the smoke detector warning system. Note that this

  does not test the smoke detectors themselves, but

  just the systems response to smoke detector signals. *

- time_of_day (store, pel) =

  * This is a read-only store indicating time of day. *.

- time_received (data flow, pel) =

  * This is the time a sensor value was received. *.

- timeout (control flow, del) =

  ["TRUE" | "FALSE"].

the status is set to out of range because this condition

is treated the same as if the sensor is out of range. *

- sensor_type (data flow, pel) =

  ["S" | "F" | "P" | "T"].

  * A unique letter identifying the type of sensor. *

- sensor_value (data flow, pel) =

  * Value of the reading from a sensor. *.

- set_timer (control flow, del) =

  ["TRUE" | "FALSE"].

- smoke_detection (control flow, del) =

  ["TRUE" | "FALSE"].

  * If true then smoke has been detected. If false

  then smoke is no longer detected. *

- smoke_detector_status (store, pel) =

  * Status of smoke detector. *.

- smoke_message (data flow, pel) =

  "WARNING !!! SMOKE DETECTED."

- state (data flow, pel) =

  ["IN_RANGE" | "OUT1" | "OUT2" | "OUT_OF_RANGE" | "IN1" | "IN2"].

  * State in which a particular sensor exists. *

- temp_message (data flow, pel) =

  "WARNING !!! ENGINE TEMPERATURE OUT OF RANGE.".

- sensor_data_received (control flow, del) =

  ["TRUE" | "FALSE"].

- sensor_id (data flow) =

  sensor_type + sensor_number.

- sensor_lamp_command (control flow, del) =

  ["RED" | "GREEN"].

- sensor_number (data flow, pel) =

  * A unique number assigned to a sensor. *.

- sensor_reading (store) =

  @sensor_id + sensor_value + @time_received.

  * It is assumed that there is at least one temperature, one

  pressure, one fuel sensor, and one smoke detector onboard. *

- sensor_state (store) =

  3{@sensor_id + state}3

  * A state is maintained to determine how many out

  of range readings have occurred for the fuel tank,

  the pressure sensor, and the temperature sensor. *

- sensor_status (control flow, del) =

  ["IN" | "OUT"].

  * Indicates that a particular sensor is either

  in range or out of range. If a sensor timed out,

* Status generated from reading sensor data from
the device itself. *

- reading_request (control flow, del) =
  ["TEMP" | "PRESS" | "FUEL"].
  * This is the signal generated to poll the different
  sensors *
- record_timeout (control flow, del) =
  ["TRUE" | "FALSE"].
- recording_buffer (store) =
  {recording_data}.
  * Formatted data for recording on magnetic medium. *
- recording_data (data flow, pel) =
  * This is the sensor data, formatted for the magnetic
  medium that is required to be recorded. *.
- request_id (data/control flow, pel) =
  * Unique identifier for a pilot request. *.
- sensor_data (data flow) =
  sensor_id + sensor_value.
  * This is the data coming in from the sensor itself. *
- sensor_data_buffer (store) =
  {sensor_data}
  * This is the data coming in from the sensor itself
  which is stored in a buffer. *

- pilot_request_buffer (store) =

  {pilot_request}.

  * Commands from the pilot. See pilot_request. *

- pilot_requested_data (data flow, pel) =

  * This information was not specified in the problem

  statement and therefore is TBD *.

- press_message (data flow, pel) =

  "WARNING !!! ENGINE PRESSURE OUT OF RANGE.".

- pressure_data (data flow, pel) =

  * Pressure returned from the engine pressure

  sensor. *.

- pressure_data_buffer (store, pel) =

  * Pressure returned from the engine pressure

  sensor. *.

- pressure_data_received (control flow, pel) =

  ["TRUE" | "FALSE"].

  * Indicates whether or not a response was received

  from the polling of the pressure sensor. *

- range_constants (store) =

  2{@sensor_id + lower_limit + upper_limit}2.

  * Contains the upper and lower limits for the engine

  pressure sensor and the engine temperature sensor. *

- read_status (control flow, pel) =

  ["OK" | "BAD"].

- fuel_message (data flow, pel) =

  "WARNING !!! FUEL CAPACITY BELOW 10 %".
- fuel_status (control flow, del) =

  ["LOW" | "OK"].
- lamp_buffer (store) =

  {@sensor_id + lamp_msg}.

  * Command to turn lamp to RED or GREEN. *
- lamp_command (control flow) =

  [fuel_lamp_command | sensor_lamp_command | detector_lamp_command].
- lamp_msg (data flow, pel) =

  * Formatted message to turn lamp RED or GREEN. *.
- lower_limit (data flow, pel) =

  * The lower limit in an allowable range

  for a sensor. *.
- one_second_interrupt (control flow, del) =

  * Interrupt generated by the clock

  every one second. This interrupt shall

  cause a polling request to all sensors. *.
- pilot_data_queue (store, pel) =

  * This is the pilot-requested data that is

  queued for display. Requirements for the

  specific types of data are TBD. *.
- pilot_request (data flow) =

  [acknowledge_alert | test_smoke_detector | calculate_data].

- dial_data_msg (data flow, pel) =

  * This is the converted sensor reading that gets sent

  to the dial. *.

- display_buffer (store) =

  {display_msg}.

  * Contains formatted messages queued for

  display. *

- display_data (data flow) =

  alert_message + pilot_requested_data.

- display_msg (data flow, pel) =

  * The formatted text for display of alerts to

  the pilot. *.

- fuel_data (data flow, pel) =

  * Fuel amount remaining returned from the fuel tank. *.

- fuel_data_buffer (store) =

  fuel_data.

  * Contains the fuel_data for the fuel tank. *

- fuel_data_received (control flow, del) =

  ["TRUE" | "FALSE"].

  * Indicates whether or not a response was received

  from the polling of the fuel tank. *

- fuel_lamp_command (control flow, del) =

  ["RED" | "GREEN"].

* This control flow indicates that all alerts have been acknowledged so that pilot requested data may be displayed. *

- calculate_data (data flow, del) =

  ["READING" | "PRESS_CHANGE" | "FUEL_CONSUMP"].

  * Note: The system level requirements did not specify the types of data to be CALCULATED from the sensor data. It provided the two examples 1) rate of change in pressure and 2) rate of fuel consumption. It gave one specific pilot request of displaying the latest recorded sensor data. Therefore, the final form of this definition is TBD. *

- detection_type (data flow, pel) =

  ["SMOKE" | "NO SMOKE"].

  * This is a data flow from the smoke detector that indicates whether smoke has been detected or whether smoke is no longer detected. *

- detector_lamp_command (control flow, del) =

  ["RED" | "GREEN"].

- dial_buffer (store) =

  {dial_data_msg}.

  * Contains formatted dial readings queued for display. *

Data Dictionary Entries

The DDEs for the AMS are listed below:

- DDEs:

- acknowledge_alert (data flow) =

  sensor_id.

- alert_acknowledged (control flow, del) =

  ["TRUE" | "FALSE"].

- alert_action (control flow, del) =

  ["GENERATE" | "RESET"].

- alert_message (data flow) =

  [press_message | temp_message | fuel_message | smoke_message].

- alert_messages (store) =

  @sensor_id + display_msg.

  * A read-only store containing the text for

  the alert messages to be displayed. *

- alert_queue (store) =

  {@sensor_id}.

  * This data store indicates which sensor alerts are

  queued for display. *

- alert_queue_status (control flow, pel) =

  ["OK" | "EMPTY"].

- all_alerts_acknowledged (control flow, del) =

  ["TRUE" | "FALSE"].

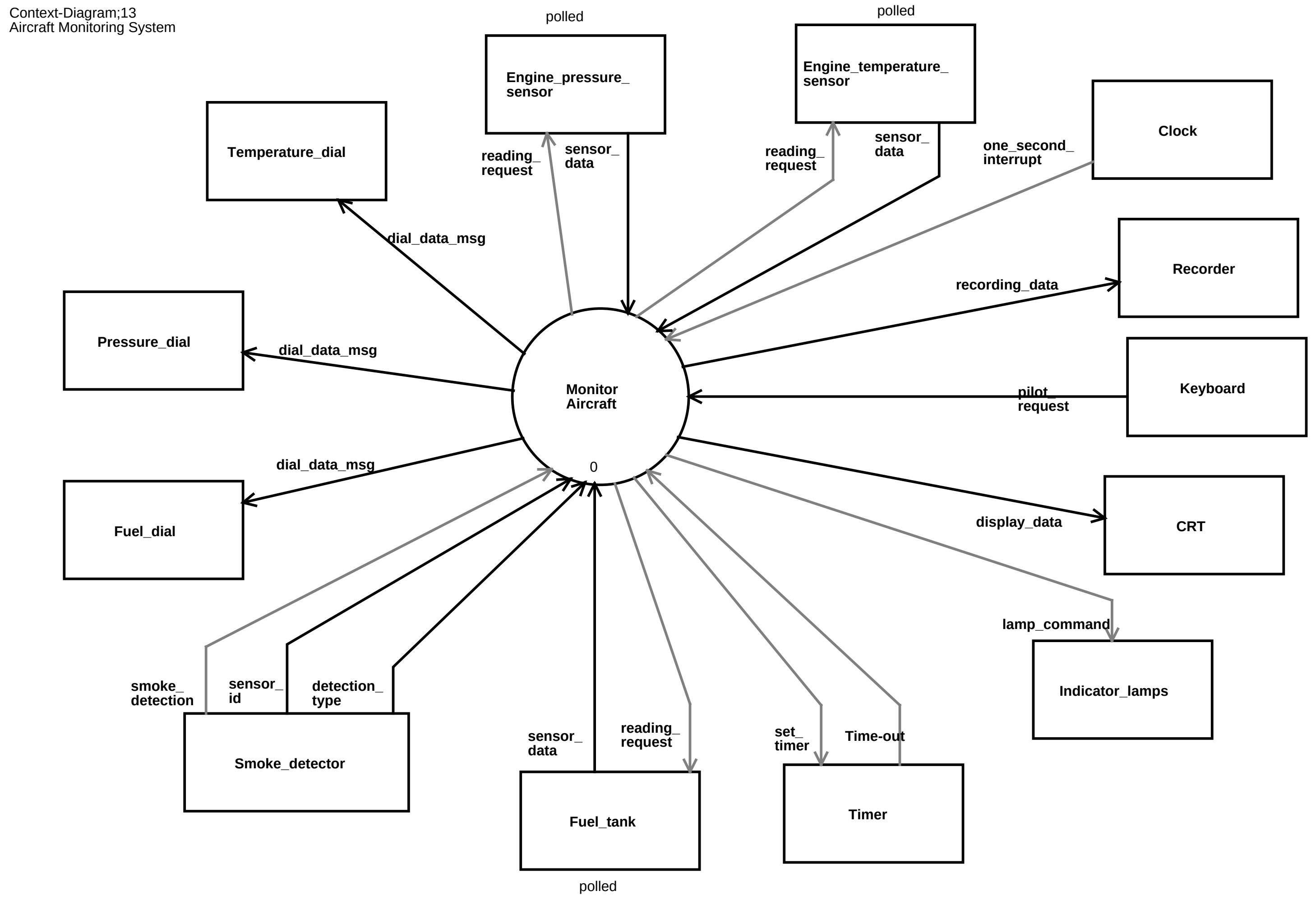


Figure 3.22 DFD Context Diagram Aircraft Monitoring System

3.2.1.3 Aircraft Monitoring System

The structured analysis example presented in this section is based on the requirements for the AMS system outlined in Chapter 1. The following list summarizes the set of diagrams and P-specs discussed in this section

•	DFD	Context-Diagram	Aircraft Monitoring System (8, 9, 10, 11, 12, 13)
•	DFD	0	Monitor Aircraft (19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34)
•	PAT	0-s1	Monitor Aircraft PAT (1, 2, 3)
•	DFD	1	Monitor Sensor
•	PAT	1-s1	Monitor Sensor PAT (1, 2, 3, 4)
•	SEM	1-s2	Monitor Sensor SEM (1, 2)
•	PS	1.1	Record Time-out (1, 2)
•	PS	1.2	Determine Range (1, 2, 3, 4)
•	PS	1.3	Determine Fuel Capacity (1, 2, 3)
•	PS	1.4	Receive Sensor Data (1, 2, 3, 4, 5)
•	DFD	4	Generate Alarm (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)
•	SEM	4-s1	Generate Alarm STD (1)
•	PAT	4-s2	Generate Alarm PAT (1, 2, 3, 4, 5, 6, 7, 8)
•	STD	4-s12	Generate Alarm STD (1, 2, 3, 4, 5)
•	PS	4.1	Add Out-of-Range Alert to Queue (1)
•	PS	4.2	Reset Lamp (1, 2, 3, 4, 5)
•	PS	4.3	Add Smoke Detector Alert to Queue (1, 2, 3)
•	PS	4.4	Add Fuel Alert to Queue (1)

Figure 3.22 shows the context diagram. According to the requirements in section 1.3.1, the Engin_pressure_sensor, Engin_temperature_sensor, and the Fuel_tank sensor are polled by the system at regular one second intervals (requirements numbers 3 and 14). The one_second_interrupt input from the clock is used by the system to start the polling sequence. Each sensor must respond with the sensor_data within a time-out interval maintained by the Timer which is set by the system for each polling activity. Sensor_data are read, and are transformed into dial_data_msg sent to the dials. Indicator_lamps are also set by the system to indicate warning conditions. The smoke_detector system sends a smoke detection signal to AMS followed by detection_type and sensor_id information. Pilot_request information is read by the system from a keyboard. This information consist of requests for measures calculated from the The system outputs to the CRT display_data containing warning messages or other information requested by the pilot such as rate of change of temperature, pressure, or fuel consumption. Finally, recording_data information consisting of all smoke or no smoke interrupts and readings tagged with time are sent to a data recoder (requirements number 25 and 26).

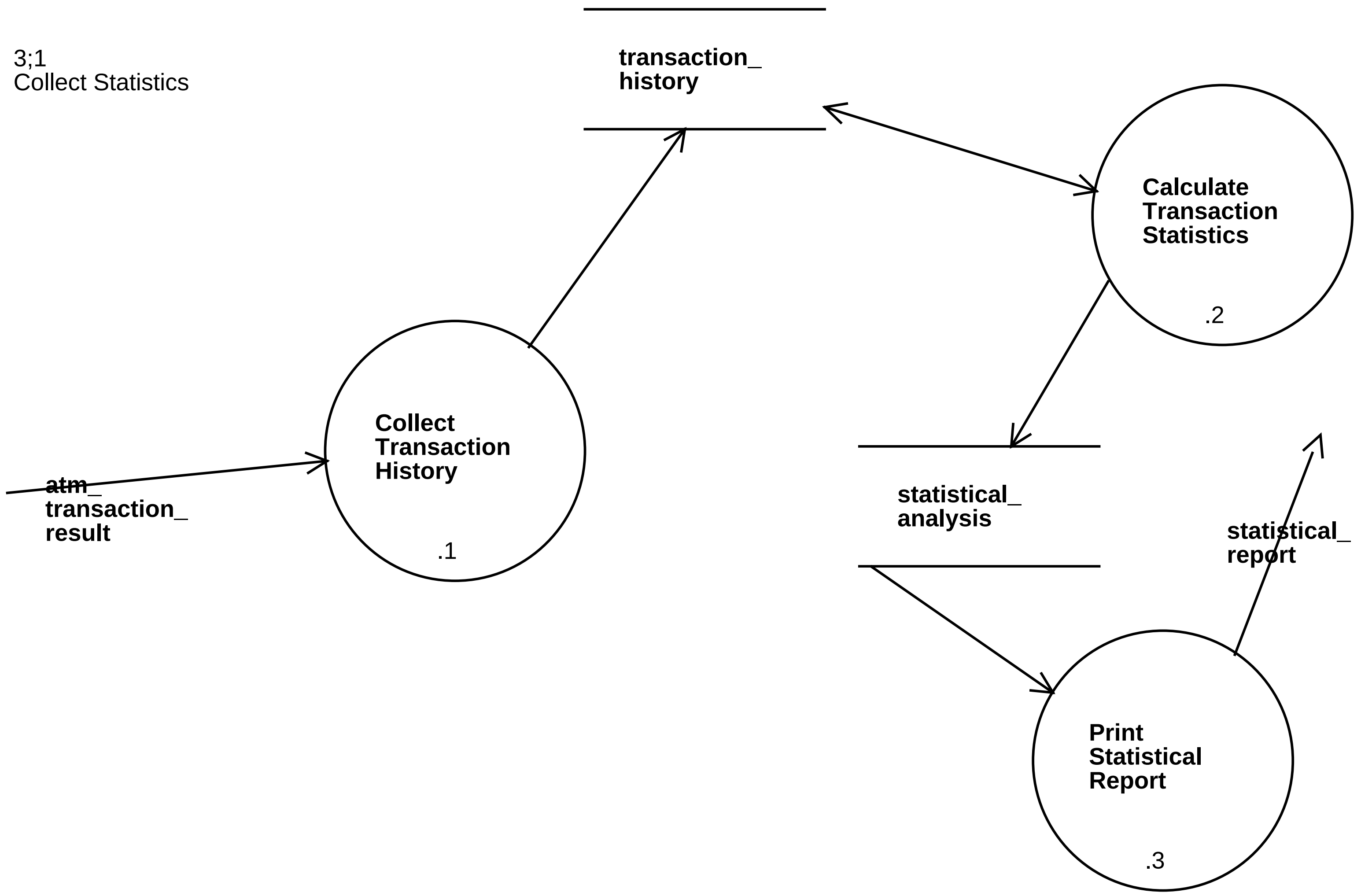


Figure 3.20 Collect Statistics DFD

Control Transactions PAT

transaction_type	**“Execute Inquiry”**	**“Execute Transfer”**	**“Execute Deposit”**	**“Execute Withdrawal”**
“WITHDRAWAL”				**1**
“INQUIRY”	**1**			
“DEPOSIT”			**1**	
“TRANSFER”		**1**		

2;4
Execute Transaction

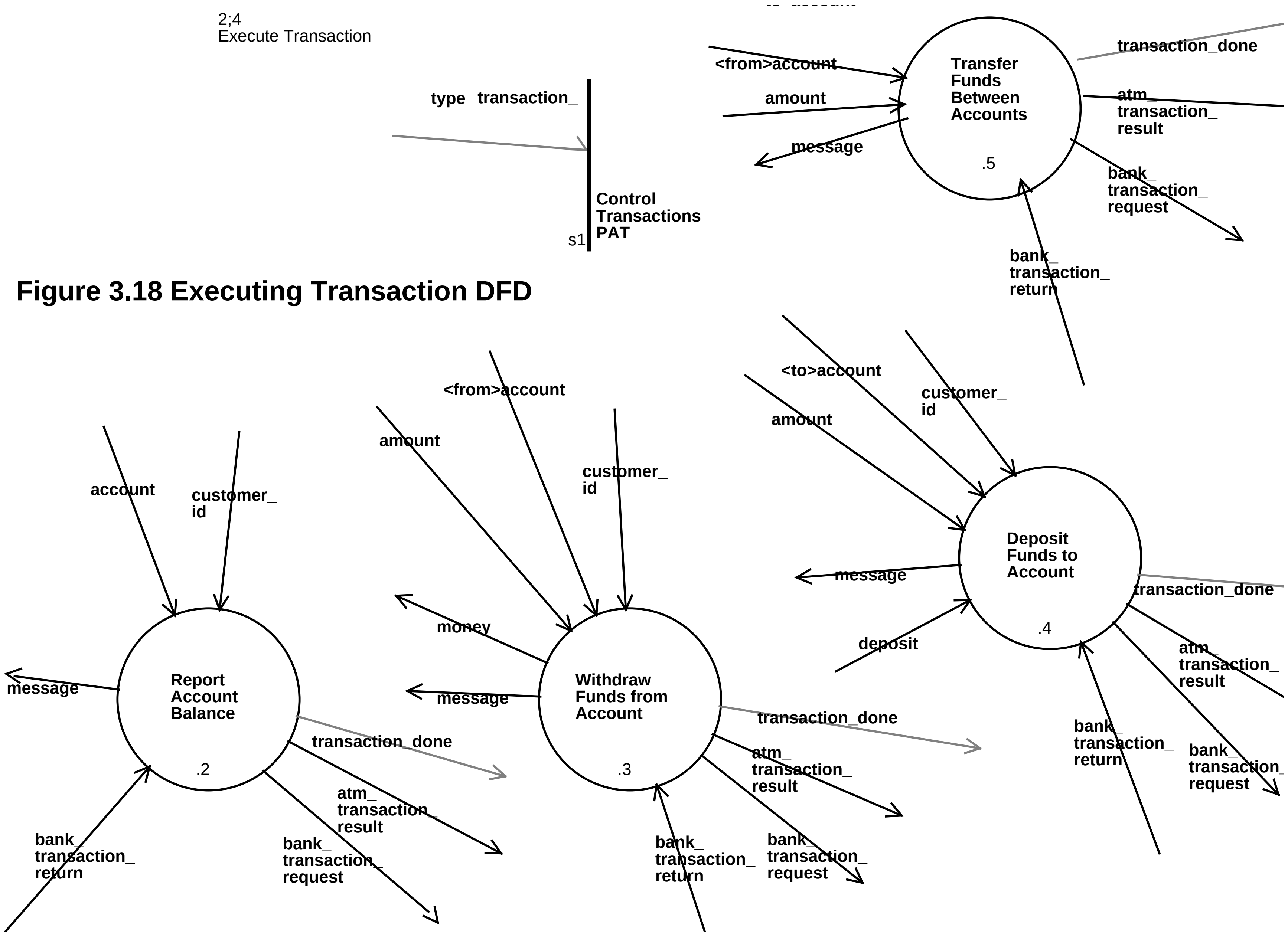


Figure 3.18 Executing Transaction DFD

3.2.1.2.1 Check Security Code

```
NAME: ATM

TITLE:
Check Security Code
      INPUT/OUTPUT:
      Customer_Card_Info: data_in
      card_identification_number: data_in
      security_code: data_in
      customer_id: data_out
      message: data_out


      BODY:
      Look up the card_identification_number in the Customer_Card_Info
      store. If the atm card's security code matches the <user_entered>
      security_code, send out the corresponding customer_id. If the
      <user_entered>security_code does not match the assigned security_code,
      send an <error>message to the Customer.
```

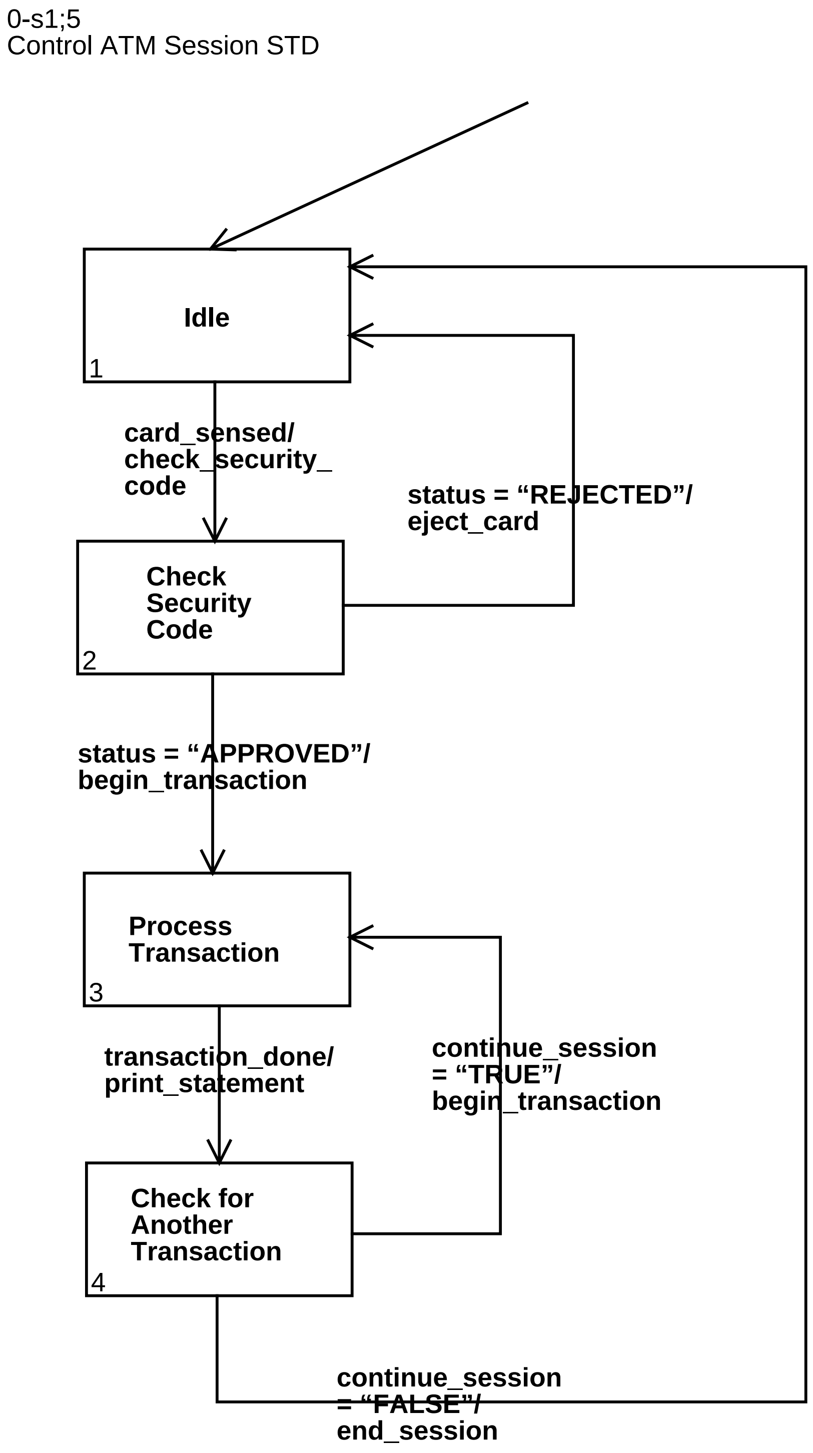


Figure 3.16 STD of a Control ATM Session

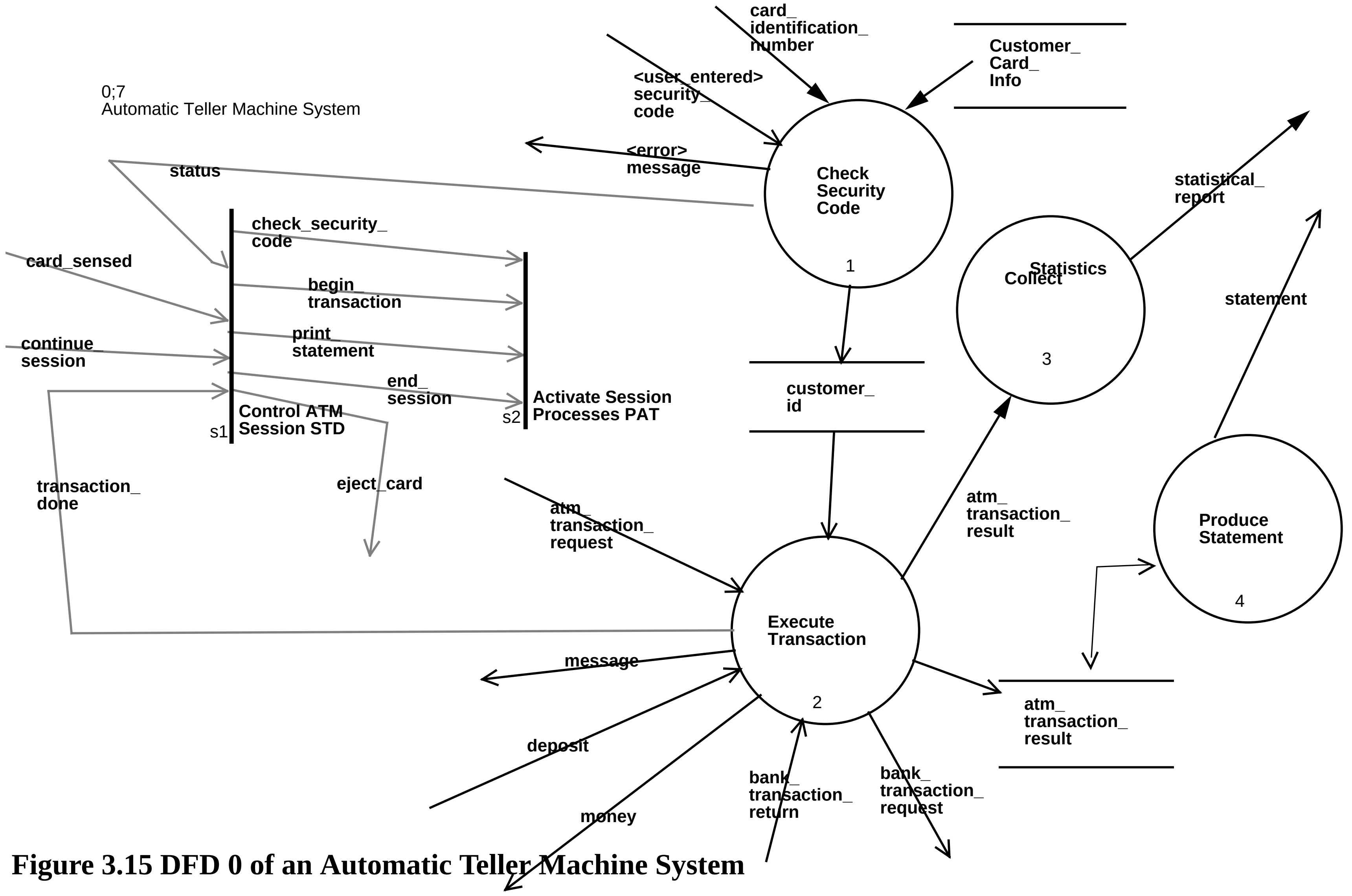


Figure 3.15 DFD 0 of an Automatic Teller Machine System

- status (control flow, del) =

  ["APPROVED" | "REJECTED"]

  * specifies whether the security code has been approved or rejected. *
- time (data flow, cel) =

  * time in hours and minutes, on the 24-hour clock *
- transaction_data (data flow) =

  [withdrawal_data | inquiry_data | deposit_data | transfer_data]
- transaction_done (control flow, del) =

  ["TRUE" | "FALSE"]
- transaction_history (store) =

  @transaction_id + atm_location + transaction_type + 1{account}2 + (amount) +

  1{balance}2 + date + time
- transaction_id (data flow, cel) =

  * a unique number identifying the particular transaction *
- transaction_type (control flow, del) =

  ["WITHDRAWAL" | "INQUIRY" | "DEPOSIT" | "TRANSFER"]
- transfer_data (data flow) =

  <from>account + <to>account + amount + customer_id
- withdrawal_data (data flow) =

  <from>account + amount + customer_id
- year (data flow, cel) =

  * the last two digits of the current year *
-

- month (data flow, pel) =

  * one of the numbers from 01 to 12, inclusive *

- print_statement (control flow, del) =

  ["TRUE" | "FALSE"]

- requested_deposits (store) =

  @transaction_id + <to>account + amount + customer_id

- requested_inquiries (store) =

  @transaction_id + customer_id + account

- requested_transfers (store) =

  @transaction_id + <from>account + <to>account + amount + customer_id

- requested_withdrawals (store) =

  @transaction_id + <from>account + amount + customer_id

- security_code (data flow, cel) =

  * a numeric code that allows customers access to

  the ATM network with their ATM card. *

- statement (data flow) =

  transaction_id + atm_location + date + time + customer_id + transaction_type +

  transaction_data + 1{balance}2

- statistical_analysis (store) =

  * statistics breaking down ATM usage by time, day of the week, location, customer

  account type, account balance, and amount of transaction. *

- statistical_report (data flow, cel) =

  * a report on ATM usage by ATM location and customer

  account type. *

Figure 3.25 Monitor Sensor PAT

- customer_id (store, cel) =

  * a ten-digit number that uniquely identifies a

    Westbrook Bank customer. *

- daily_withdrawal_record (store) =

  @customer_id + <todays_withdrawal>amount

- date (data flow) =

  month + day + year

- day (data flow, del) =

  * one of the numbers from 01 to 31, inclusive *

- deposit (data flow, del) =

  ["TRUE" | "NULL"]

  * receipt of an ATM deposit envelope from the customer. *

- deposit_data (data flow) =

  <to>account + amount + customer_id

- end_session (control flow, del) =

  ["TRUE" | "FALSE"]

- inquiry_data (data flow) =

  account + customer_id

- insufficient_funds (data flow, del) =

  ["TRUE" | "FALSE"]

- message (data flow, del) =

  * an error message or prompt to be displayed to the customer. *

- money (data flow, cel) =

  * an amount of cash for the teller machine to give to the customer. *

- bank_transaction_result (data flow) =

  transaction_id + transaction_type + balance + (<second>balance)
- bank_transaction_return (data flow) =

  [bank_transaction_result | bank_transaction_error]

- begin_transaction (control flow, del) =

  ["TRUE" | "FALSE"]
- card_identification_number (data flow, cel) =

  * a 12-digit code that uniquely identifies a

    particular ATM card. *
- card_sensed (control flow, del) =

  ["TRUE" | "FALSE"]

  * a signal specifying that an atm card has been inserted

    into the ATM *
- check_security_code (control flow, del) =

  ["TRUE" | "FALSE"]
- continue_session (control flow, del) =

  ["TRUE" | "FALSE"]

  * the customer specifies whether or not s/he wants to

    continue the ATM session *
- Customer_Card_Info (store) =

  security_code + @card_identification_number + customer_id

Figure 3.25 Monitor Sensor SEM

Data Dictionary Entries for the ATM system

The DDEs are listed as given below:

- account (data flow, del) =

  ["NOW" | "Money_Market" | "Checking" | "Savings"]

- account_doesnt_exist (data flow, del) =

  ["TRUE" | "FALSE"]

- amount (data flow, cel) =

  * the amount of money involved in a transaction;

    consists of any number of digits followed by

    a decimal point, followed by two digits *

- atm_location (data flow, del) =

  * the branch location of the automatic teller machine *

- atm_transaction_request (data flow) =

  transaction_type + transaction_data

- atm_transaction_result (data flow) =

  transaction_id + customer_id + atm_location + transaction_type +

  transaction_data + balance + (<second>balance)

- balance (data flow, cel) =

  * the amount of money currently available in the customer's account *

- bank_transaction_error (data flow) =

  transaction_id + transaction_type + [account_doesnt_exist | insufficient_funds]

- bank_transaction_request (data flow) =

  customer_id + transaction_id + transaction_type + transaction_data

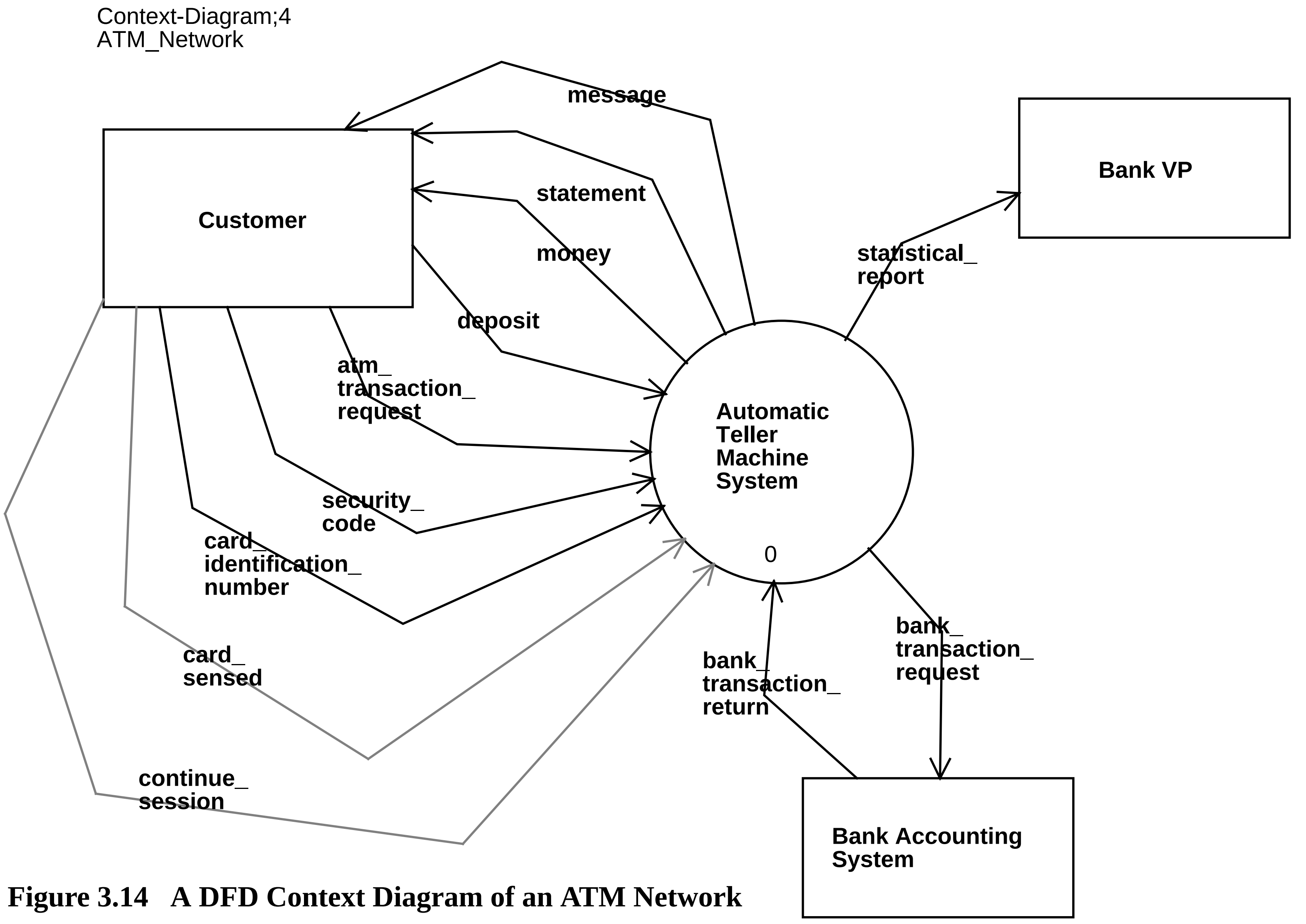


Figure 3.14 A DFD Context Diagram of an ATM Network

3.2.3.2 The ATM Example

This section presents the specification of a typical ATM example. The DFD 0 for this example has already been discussed in the notation description section. The context diagram is shown in the next page in Figure 3.14. This diagram shows the typical inputs and outputs of an ATM system. The system inputs and outputs to/ from the customer are straightforward. A timing deadlines in terms of few seconds should specified for the outputs to the customer. The ATM interacts with a bank accounting system and a bank statistics gathering system

DFD 0 is shown again in Figure 3.15. The structure of DFD 0 is typical of many transaction processing systems. The input data processing node, node 1, deals with input information items such as the card_identification_number. The transaction execution node deals with inputs selecting the transaction type and specifying transaction information, and produces transaction result information used by the two output information producing nodes. The control is divided into two nodes S1 and S2. S1 reads input control signals and uses them to produce the control signals needed to manage an ATM session. S2 receives control signals from S1 to activate and deactivate the data processing nodes

The following two figures Figure 3.16 and Figure 3.17 give the C-specs for controllers S1 and S2. The STD in Figure 17 for S1 specifies the states of the system and the events triggering a state change. The PAT in Figure 3.16 shows the activation conditions for the data processing nodes in DFD 0.

The P-spec for node 1 in DFD 0 is shown in Figure 3.18. The execute_transaction node, node 2 in DFD 0, is complex enough and needs to be specified further by a lower level DFD. Figure 3.19 shows the decomposition of this process in DFD 2. The data processing nodes in this DFD are divided according to transaction type (such as withdraw, transfer, deposit, etc.). The controller S1 activates one of these processes according to the transaction_type input control. The C-spec of S1 is shown in Figure 3.20.

Finally Figure 3.21 shows DFD 3, the lower-level DFD diagram for node 3 in DFD 0. This node specifies the collect statistics data processing functions.The following list summarizes the set of ATM diagrams shown in the next few pages These diagrams are not a complete specification for the system since several P-specs of simple function were omitted. Also the data dictionary entries for the data flow, control flow, and stores must be specified. This list appears in the process index of Teamwork for the ATM model. The number associated with the diagram name in the list specifies the number of revisions the diagram went through.

•	DFD	Context-Diagram	ATM_Network
•	DFD	0	Automatic Teller Machine System
•	STD	0-s1	Control ATM Session STD
•	PAT	0-s2	Activate Session Processes PAT
•	PS	1	Check Security Code
•	DFD	2	Execute Transaction (1, 2, 3, 4)
•	PAT	2-s1	Control Transactions PAT (1)
•	DFD	3	Collect Statistics (1)

```
Logical_Traffic_Signal_Id: data_in
BODY:

For each entry in Low_Traffic_Flashing_Definition
(i.e. each Logical_Traffic_Lane (identified by Street_Id, Direction, and Logi-
cal_Traffic_Lane_Id))

 begin
    Determine Logical_Traffic_Signal_Id from Logical_Traffic_Lanes store
    If Traffic_Signal_State =
             Red  : Generate FLASH_RED
                      command to the signal indicated by the Logical_Traffic_-
Signal_Id
                     ...
   end
```

Figure 3.13 (c)

(c) Flash_ Out_of_Order P-Spec

```
NAME: Traffic Light System
4
TITLE:
Flash_ Out_of_Order
INPUT/OUTPUT:
Out_of_Order_Flashing_Definition: data_in
Traffic_Light_Commands: data_out
Logical_Traffic_Signal_Id: data_in
BODY:

For each entry in Out_of_Order_Flashing_Definition
(i.e. each Logical_Traffic_Lane (identified by Street_Id, Direction, and Logi-
cal_Traffic_Lane_Id))

 begin
    Determine Logical_Traffic_Signal_Id from Logical_Traffic_Lanes store
    If Traffic_Signal_State =
             Red  : Generate FLASH_RED
                      command to the signal indicated by the Logical_Traffic_-
Signal_Id
             Yellow: Generate FLASH_YELLOW command
                     ...
   end
```

Figure 3.13 (a)

(a) Flow Schedule P-Spec

```
NAME: Traffic Light System
2
TITLE:
Follow_ Schedule
INPUT/OUTPUT:
Change_Mode: control_out
Daily_Schedule: data_in
Current_Mode: data_inout
Time_of_Day: data_in
      BODY:

      if Current_Mode is Signal_Control then
         if Time_of_Day is equal to or later than
      Daily_Schedule.Flash_Low_Traffic_Start_Time
           then
                 generate Change_Mode event (i.e. set Change_Mode to TRUE)
                 done
      endif

      if Current_Mode is Flashing_Low_Traffic then
         if Time_of_Day is equal to or later than
      Daily_Schedule.Control_Intersection_Cycle_Start_Time
           then
                 generate Change_Mode event (i.e. set Change_Mode to TRUE and
      send)
                 done
      endif
```

Figure 3.13 (b)‘

(b) Flash Low Traffic P-Spec

```
NAME: Traffic Light System
3
TITLE:
Flash_ Low_Traffic
INPUT/OUTPUT:
Low_Traffic_Flashing_Definition: data_in
Traffic_Light_Commands: data_out
```

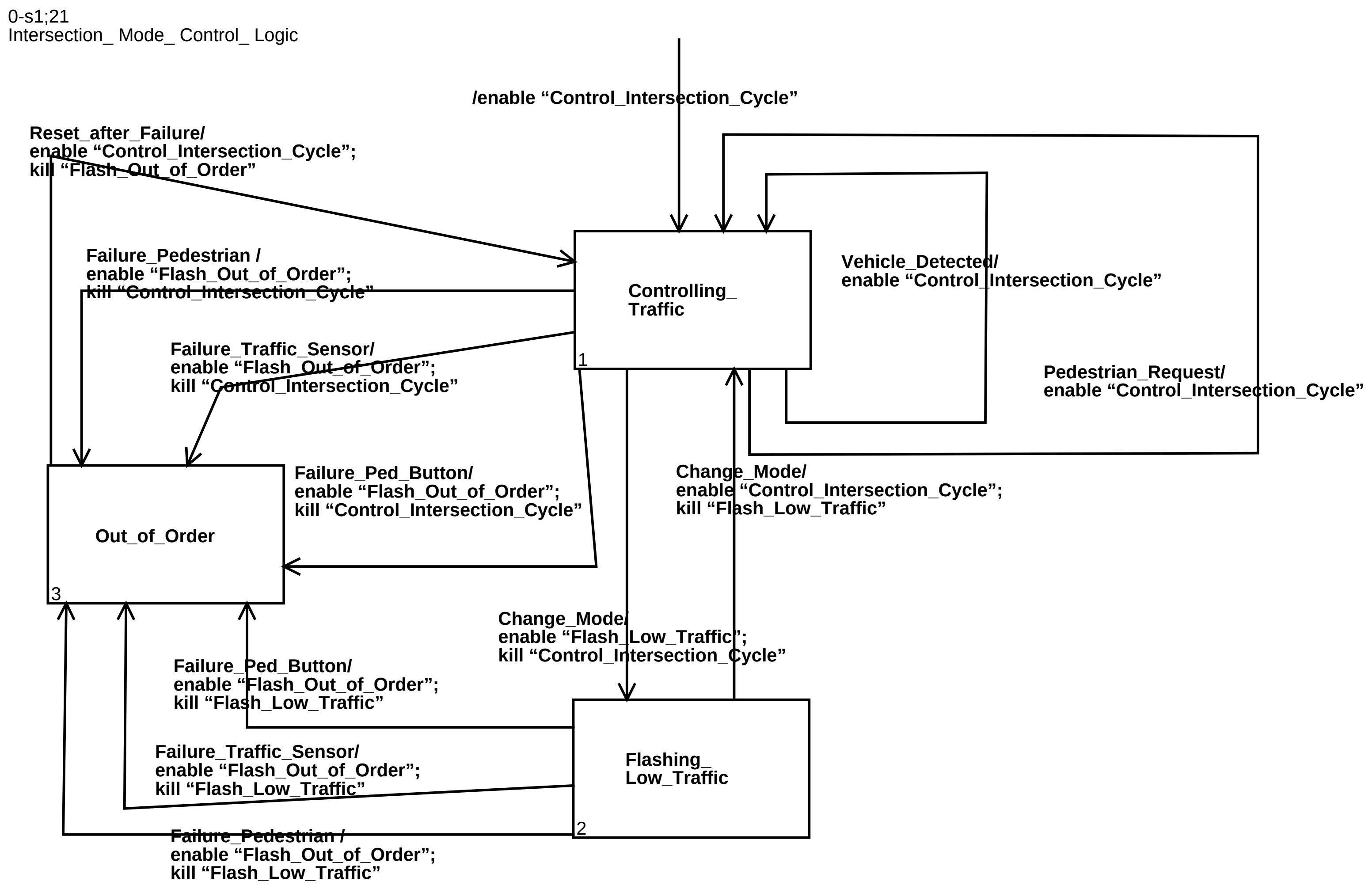

0-s1;21
Intersection_ Mode_ Control_ Logic
/enable “Control_Intersection_Cycle”
Reset_after_Failure/
enable “Control_Intersection_Cycle”;
kill “Flash_Out_of_Order”
Failure_Pedestrian /
enable “Flash_Out_of_Order”;
kill “Control_Intersection_Cycle”
Controlling_
Traffic
1
Vehicle_Detected/
enable “Control_Intersection_Cycle”
Failure_Traffic_Sensor/
enable “Flash_Out_of_Order”;
kill “Control_Intersection_Cycle”
Pedestrian_Request/
enable “Control_Intersection_Cycle”
Failure_Ped_Button/
enable “Flash_Out_of_Order”;
kill “Control_Intersection_Cycle”
Change_Mode/
enable “Control_Intersection_Cycle”;
kill “Flash_Low_Traffic”
Out_of_Order
3
Change_Mode/
enable “Flash_Low_Traffic”;
kill “Control_Intersection_Cycle”
Failure_Ped_Button/
enable “Flash_Out_of_Order”;
kill “Flash_Low_Traffic”
Failure_Traffic_Sensor/
enable “Flash_Out_of_Order”;
kill “Flash_Low_Traffic”
Flashing_
Low_Traffic
2
Failure_Pedestrian /
enable “Flash_Out_of_Order”;
kill “Flash_Low_Traffic”

Figure 3.12 The C-Spec sheet of the controller in DFD/CFD 0

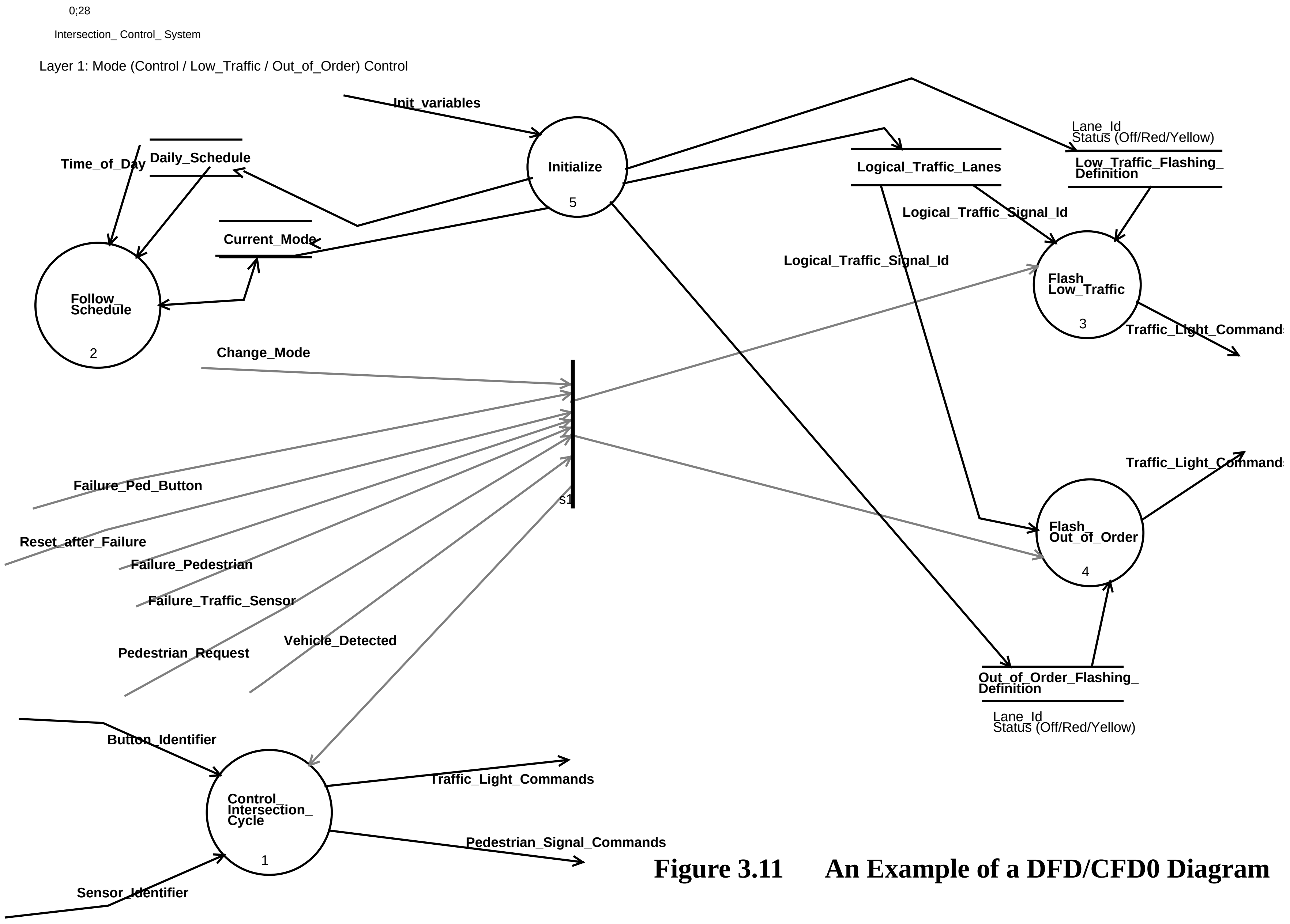


Figure 3.11 An Example of a DFD/CFD0 Diagram

Figure 3.19 Control Transactions PAT

The controller bar shown in the middle Figure 3.11 activates or deactivates the functions 1, 3, and 4. The unlabeled control flow signals originating at the controller bar to these functions are actually activation signals and should not be shown in the figure. A syntax error will be produced by the Check facility of Teamwork/RT when the syntax of the diagram is being checked. The C-spec for this controller is shown in Figure 3.12. This is a Mealy STD with three states. The initial state Controlling_Traffic represents the normal mode of operation for the system. Any transition entering this state activates the Control_Intersection_Cycle process in Figure 3.11 (bubble 1) which generates the proper commands for this mode.

Process Specifications

Primitive processes in DFD/CFD 0 are represented by P-specs as shown below in Figure 3.13. The specifications of processes 2, 3, and 4 are given. The name, title, and the input/output section of each process will be automatically generated by Teamwork/RT when the P-specs are created for the respective bubbles from DFD/CFD 0. The body section is specified by the analyst to show how the output flows are obtained from the input flows. The specification body shown for process 2 in Figure 3.13 (a) is incomplete, it shows that a value of TRUE is given to the output flow Change_Mode when a condition for changing the operation is satisfied. However, the value of the Current_Mode store is not changed to reflect the new current mode of operation. The specification should be completed by adding a statement to set the current mode to the proper value when a mode change occurs. The specification of processes 3 and 4 show the commands generated for the low traffic or out-of-order flashing modes of operations, respectively.

References

[TWK/RT] Teamwork/RT User’s guide, Cadre technologies, Inc.

[DEMARCO 82] Tom DeMarco, Controlling Software Projects, Yourdon Press, 1982.

- Time_of_Day (data flow, pel) =

  * HH : MM : SS in 24 hour format *

- Traffic_Light_Commands (data flow, pel) =

  Logical_Traffic_Signal_Id

  + Next_State_Traffic_Signal

- Traffic_Signal_State (data flow, del) =

  ["RED" | "YELLOW"]

- Trigger_Control_Intersection_Cycle (control flow, del) =

  ["TRUE" | "FALSE"]

- Trigger_Flash_Low_Traffic (control flow, del) =

  ["TRUE" | "FALSE"]

- Trigger_Flash_Out_of_Order (control flow, del) =

  ["TRUE" | "FALSE"]

- Type (data flow, pel) =

  ["Left_Turn" | "Through"]

- Vehicle_Detected (control flow, del) =

  "TRUE"

DFD/CFD 0 and Control Specifications C-spec 0-s1

The top level functional decomposition shown in Figure 3.11 consists of five functions. The functions are briefly described as follows. An Initialize function used to initialize the system parameters. These parameters are stored in five data stores specifying respectively the daily schedule of operation store, the current mode of operation store, the specifics of the traffic lanes of the intersection store, low traffic flashing definition store (e.g., flashing yellow signal or flashing red signal for each lane), and the out-of-order flashing definition store. The Follow_Schedule function changes the mode of operation of the system depending on the command flows for the three different modes of operations respectively.

Figure 3.17 Activate Session Processes PAT

2;1

Activate Session Processes PAT

check_security_code	begin_transaction	print_statement	end_session	“Check Security Code”	“Execute Transaction”	“Collect Statistics”	“Produce Statement”
“TRUE”	*can't happen*	*can't happen*	*can't happen*	1	0	0	0
can't happen	“TRUE”	*can't happen*	*can't happen*	0	1	0	0
can't happen	*can't happen*	“TRUE”	*can't happen*	0	0	1	0
can't happen	*can't happen*	*can't happen*	“TRUE”	0	0	0	1

- Pedestrian_Lane_Id (data flow, pel) =

  * integer - assigned as an identifier for each

    cross walk or pedestrian lane. *

- Pedestrian_Lanes (store, del) =

  0 { Pedestrian_Lane_Id

  + Crossing_Street_Id

  + Direction

  + Logical_Pedestrian_Sensor_Id

  + Logical_Pedestrian_Signal_Id }

- Pedestrian_Request (control flow, del) =

  "TRUE"

- Pedestrian_Signal_Commands (data flow, pel) =

  Logical_Pedestrian_Signal_Id + Next_State_Ped_Signal

- Reset_after_Failure (control flow, del) =

  "TRUE"

- Sensor_Identifier (data flow, pel) =

  * reference to table containing physical interface information. *

- Signal_Control (data flow) =

  not-defined.

- Street_Id (data flow, pel) =

  *alpha_numeric - an assigned key or identifier *

  *refers to the entries in the Intersection Definition table. *

- Logical_Traffic_Signal_Id (data flow) =

  * Reference to physical interface information. *

- Low_Traffic_Flashing_Definition (store) =

  { Street_Id

    + Direction

    + Logical_Traffic_Lane_Id

    + Traffic_Signal_State

  }.

- Next_State_Ped_Signal (data flow, pel) =

  ["WALK" | "FLASHING_WAIT" | "WAIT"]

  *This actually "implementation specific". These values are exemplary. *

- Next_State_Traffic_Signal (data flow, pel) =

  not-defined.

- Out_of_Order_Flashing_Definition (store) =

  {

    Street_Id + Direction + Logical_Traffic_Lane_Id

    + Traffic_Signal_State

  }.

- Init_variables (data flow, pel) =

  not-defined.

- Logical_Pedestrian_Sensor_Id (data flow, pel) =

  * reference to table contiaining interface information for

    these devices *

- Logical_Pedestrian_Signal_Id (data flow, pel) =

  * reference to a table contining the physical interface

    interface information for these devices *

- Logical_Traffic_Lane_Id (data flow, pel) =

  * integer - combines with Street_Id & Direction

    to uniquely identify a logical lane *

  * one logical lane may correspond to two adjacent physical lanes*

- Logical_Traffic_Lanes (store) =

  { Street_Id

  + Direction

  + Logical_Traffic_Lane_Id

  + Type

  + Logical_Traffic_Sensor_Id

  + Logical_Traffic_Signal_Id }.

- Logical_Traffic_Sensor_Id (data flow, pel) =

  * reference to table containing physical interface information. *

* an attribute of a street *

* Each 2 way street that is part of an intersection is
divided into a primary and secondary component. For example,
given a north/south street, the north bound traffic may
be identified as the primary component, and the south identified
as the secondary component. These distinctions are seen in the
decision table - 1.5.1-s1 - the rules governing the operation
of the signal. *

* may combines with Street_Id to uniquely identify a step
of the intersection cycle *

- Failure_Ped_Button (control flow, del) =
  ["TRUE" | "FALSE"]
- Failure_Pedestrian (control flow, del) =
  ["TRUE" | "FALSE"]
- Failure_Traffic_Sensor (control flow, del) =
  ["TRUE" | "FALSE"]
- Flash_Low_Traffic_Start_Time (data flow, pel) =
  Time_of_Day
- Flashing_Low_Traffic (data flow) =
  not-defined.

Data Dictionary Entries

The DDEs for traffic light system are given below:

- Button_Identifier (data flow, pel) =

  * reference to table contiaining interface information for

    these devices *

- Change_Mode (control flow, del) =

  "TRUE"

- cont_temp (control flow) =

  { sss | aaa }

- Control_Intersection_Cycle_Start_Time (data flow, del) =

  Time_of_Day

- Crossing_Street_Id (data flow, pel) =

  * The Street_Id of the street that this pedestrian lane crosses *

- Current_Mode (store, pel) =

  ["Signal_Control" | "Flashing_Low_Traffic"]

- Daily_Schedule (store) =

  { Control_Intersection_Cycle_Start_Time

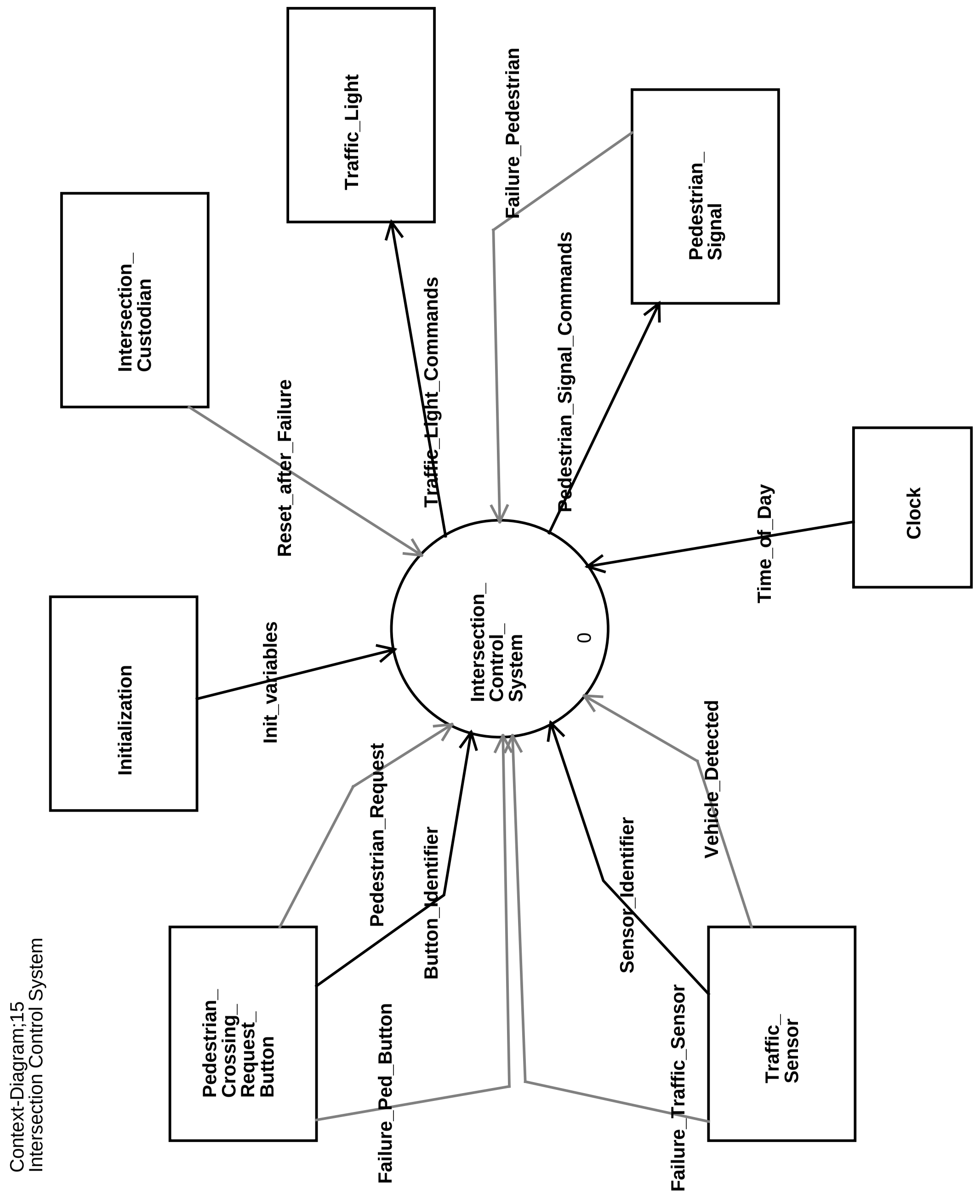
    + Flash_Low_Traffic_Start_Time

  }.

- Direction (data flow, pel) =

  ["Primary" | "Secondary"]

FIGURE 3.10 The Context Diagram Figure 3.10



intersections involving two two-way streets with left turn lanes. The following diagrams are developed using the requirements stated in Chapter 1.

The Context Diagram

The context diagram shown on the next page specifies the input and output flows and the external components interacting with the system. The system reads in sensors vehicles in the intersection or pedestrian requests and produces commands to the traffic light component and the pedestrian signal component. The system also reads in the Init_variables flow and use it to initialize the system parameters used for the different modes of operation (such as flashing signal for low traffic times, flashing out of order message, or the normal Green-Yellow-Red operation cycle). The failure signals obtained from the external devices are used to set the system in an Out_of_Order flashing mode. The Time_of_Day input is used to put the system in the Flashing_Low_Traffic mode.

•

PS	3.1	Collect Transaction History (1)
PS	3.2	Calculate Transaction Statistics (1)
PS	3.3	Print Statistical Report (1)
PS	4	Produce Statement (1)

3.2.3.3 AMS

DFD	Context-Diagram	Aircraft Monitoring System (8, 9, 10, 11, 12, 13)
DFD	0	Monitor Aircraft (19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34)
PAT	0-s1	Monitor Aircraft PAT (1, 2, 3)
DFD	1	Monitor Sensor
PAT	1-s1	Monitor Sensor PAT (1, 2, 3, 4)
SEM	1-s2	Monitor Sensor SEM (1, 2)
PS	1.1	Record Time-out (1, 2)
PS	1.2	Determine Range (1, 2, 3, 4)
PS	1.3	Determine Fuel Capacity (1, 2, 3)
PS	1.4	Receive Sensor Data (1, 2, 3, 4, 5)
DFD	4	Generate Alarm (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)
SEM	4-s1	Generate Alarm STD (1)
PAT	4-s2	Generate Alarm PAT (1, 2, 3, 4, 5, 6, 7, 8)
STD	4-s12	Generate Alarm STD (1, 2, 3, 4, 5)
PS	4.1	Add Out-of-Range Alert to Queue (1, 2, 3, 4, 5, 6, 7, 8)
PS	4.2	Reset Lamp (1, 2, 3, 4, 5)
PS	4.3	Add Smoke Detector Alert to Queue (1, 2, 3)
PS	4.4	Add Fuel Alert to Queue (1)

The first example is simple enough to grasp and yet contains all the elements and characteristics of real-time systems. The second example is somewhat more complex since it involves customer interaction. This system is used by all of us and thus is very intuitive to follow and understand. The third example is chosen to be more involved and is a good example of a hard real-time system. The requirements models presented for these systems are logical functional models of the system and do not imply any specific physical implementation.

3.2.3.1 Traffic Intersection Control System

This example is typical of many traffic intersection control systems used in any intersection involving pedestrian crossing. The system can be used with simple intersections involving one way street interrupted by crosswalk. It can also be used with typical more complex

The Teamwork Update_bang utility assists the analyst in specifying the model, the sub-tree within a model, and the above parameters for the bang measure. The Calc_bang utility calculates the bang metric and produces a table of showing the bang metric for each FPO and the total bang metric for the sub-tree.

3.2.3 Requirements Analysis Examples

In this section, several examples on the requirements analysis and specification concepts presented in the previous sections are discussed. These examples are obtained from the Teamwork Samples directory which contains examples of models developed using Teamwork tools.

The section describes some artifacts of three examples. The requirements of some of these examples are given in Chapter 1. These examples are:

1. Traffic intersection control system
2. Automatic Teller Machine system
3. Aircraft Monitoring System

DFD	Context-Diagram	Intersection Control System (15)
DFD	0	Intersection_ Control_ System (28)
STD	0-s1	Intersection_ Mode_ Control_ Logic (21)
PAT	0-s2	Process Activation Logic - Modes of Operation for Traffic Light (3)
PS	1	Control_ Intersection_ Cycle (14)
PS	2	Follow_ Schedule (36)
PS	3	Flash_ Low_Traffic (24)
PS	4	Flash_ Out_of_Order (21)
PS	5	Initialize (4)

3.2.3.2 ATM

DFD	Context-Diagram	ATM_Network
DFD	0	Automatic Teller Machine System
STD	0-s1	Control ATM Session STD
PAT	0-s2	Activate Session Processes PAT
PS	1	Check Security Code
DFD	2	Execute Transaction (1, 2, 3, 4)
PAT	2-s1	Control Transactions PAT (1)
DFD	3	Collect Statistics (1)

1. Each input or output value of a flow must be consistent with the flow DDE specification.

2. Each title or number of an activated process in a PAT, STD, or SEM must be consistent with its specification in the DFD/CFD.

3. Each column in a PAT must have a valid process title or number, and must be activated in at least one cell in this column.

4. In a STD the following specific rules are checked:

   a) Each state in a STD must have a unique name and number.

   b) There must only one initial transition, and it may only have an initial action.

   c) Each state must have at least one incoming transition.

   d) Each transition except the initial transition must have a triggering input event

5. In a SEM the following specific rules are checked:

   a) Every input event must have a designated column in the matrix with the event name appearing at the top.

   b) A state name specified as a next state in an event column must have a single row in the matrix (i.e., its name appears in the state name column).

   c) Every cell in an event column must have an action/next state or only a next state transition

The Bang Metric Utilities

The bang metric is a relative measure of the size and complexity of the systems requirements. The bang measures can be used to predict project effort and which functions will require the most resources to develop and how changing requirements will

affect the project schedule and resources allocated.

The bang metric supported by Teamwork for requirements specification artifacts is based on DeMarco’s concept of functional primitives [DEMARCO 82]. Functional primitives (FPs) are the most primitive components in the specification. In a Teamwork/RT based specification, the FPs are the primitive processes in a DFD specified by P-specs, and the Controllers in a CFD specified by a PAT, a DT, a STD, or a SEM. The bang metric is produced by assigning weights to all FPs depending on their level of complexity. FPs are classified by DeMarco according to their level of complexity. Classes of FPs such as combining input data, performing simple calculations, performing complex math computations, performing device management, displaying information, performing control and synchronization operations, etc., are given default weighting factors specified in a weight table. The analyst must specify the class of each FP in the model or sub-tree for which a bang metric is to be calculated. The analyst must also assign token count to each incoming or outgoing data flow or control flow in each FP.

4. Index editors: these are a model index editor containing a list of all models in the database, a process index editor containing a list of all objects created for a specific model, a note index containing a list of all the notes created for a specific model, and a data dictionary index editor which provides a list of all data dictionary entries (DDEs) created for a specific model.

Two important facilities which greatly solidify the role of ICASE environments in the development process are the check facility and the metric utilities. These are discussed further in the following paragraphs.

The Completeness and Consistency Check Facility

The most important facility provided by Teamwork/RT is the checking facility which checks the completeness and consistency of models and all objects created by the tool. This is crucial specially for large scale models where errors can be very difficult to trace and correct. The checking facility consists of a DFD/CFD check, a P-spec check, a Matrix check, and a STD check.

The DFD/CFD check may be applied to the specific DFD/CFD diagram where the check option is selected or to the sub-tree rooted at the current DFD/CFD. In the later case the DFD/CFD syntax and balancing of all child processes and C-specs are performed. The syntax and balancing rules checked include the following:

1. Every bubble, C-spec bar, store, and a terminator must have a name.
2. Every flow should be connected to bubble or a C-spec bar
3. Stores cannot appear in a context diagram, and terminators can only appear in a context diagram
4. There only one bubble numbered 0 in the context diagram
5. The input and output flows of a C-spec bar must be control flows
6. Every bubble must have a child DFD or a P-spec, and every C-spec bar must have a C-spec sheet.
7. Every flow on a child DFD must be appear or be an element of a flow on the parent bubble.
8. Every flow or a store must have a DDE, and every element of a compound DDE must also be defined in a DDE.
9. Input flows to a bubble represented by a P-spec can only be data flow (i.e., no control flows are allowed as inputs to a P-spec)

The C-spec Check is applied to a specific C-spec sheet to check the syntax rules for PATs, DTs, STDs, or SEMs. These rules include the following:

5. The total number of nodes in a DFD diagram should be balanced in order to keep the diagram easy to comprehend and in the same time contain enough number of nodes to reduce the number and complexity of the lower level diagrams needed in the hierarchy.
6. If there is ever a doubt about whether a flow or a store is data or control, the deciding factor depends on whether the information is used to control the execution or to compute data in the destination process. It is possible that an information item become a hybrid item when it is used for control in one process and for computation in another. In this case it is both control/data.

3.2.2 Structured Analysis Using Teamwork/RT

This section briefly describes the main features of Teamwork/RT as an example of ICASE support for structured analysis of real-time systems. Teamwork/RT is a multi-user and multi-tasking structured analysis environment. Multi-tasking in this context means that several development tasks for the objects of a given project model can be carried out concurrently by a team of analysts. The teamwork/RT tool support consists of the following editors used to create and navigate between objects of a given model:

1. DFD/CFD editor, used to develop data flow and control flow diagrams,
2. P-spec editor, used to develop and refine process specifications for primitive processes
3. Matrix editor used to specify PATs, DTs, and SEMs for control specifications (or C-specs), and
4. STD editor, used to develop Mealy or Moore state transition diagrams for C-specs.

Teamwork/RT interacts with the project database which contains all the models created in the Teamwork environment. This is supported by a set of editors which are common to other integrated tools in the ICASE environment. These editors are briefly described as follows:

. 1. Data dictionary entry (DDE) editor used to specify the definitions of data flows and stores. The DDEs created are also used an refined by design tools such as Teamwork/SD to be discussed in Chapter 4.

2. Text note editor used to specify text attached to a model or a model object. This text can specify special comments to be attached to the model or the object.

3.Picture note editor used to specify graphics to be attached to a model or a model object.

condition for grouping a set of functional requirements under one process (a set of cohesive functions should belong to the same process).

b) functions that have strong interconnections such as having access to common data stores or a large number of direct data/control flows (if these functions were defined as different processes a large number of data/control flows would couple them as well as common resources such as data/control stores and control signals coming from common controllers). This condition for defining the boundaries of a process is called the coupling condition (tightly coupled functions should belong to the same process).

c) functions that occur at the same time or with the same frequency as a consequence of a set of events. This is a weak form of coupling (compared to the one in b) above) between functions that should be grouped into one process.

d) functions that have common input and/or outputs to the same external entities (i.e, terminators).

2. Decomposition. Define the set of processes in a DFD which specify a higher level process using the following guidelines:

a) Partition a process to lower level processes in such a way that tends to minimize the interconnections (in terms of direct data/control flows) between them. This is also related to the coupling condition, mentioned above in 1. b), as it seeks to decompose a process into a set of weekly coupled processes.

b) Partition a process into lower level processes such that each of these processes have a well defined task. This guideline is related to the cohesion condition in 1. a) above.

c) Define lower level processes needed to input, monitor, or consume and validate the input data flows specified for the higher level process (these processes usually input the raw data, do some initial processing of filtering for the needed information, and also validate the format and values of these information. For example, inputting data for a set of sensors then applying data conditioning and calibration, and validating that the data are in their predefined ranges to insure that the sensors are operating correctly under no faulty conditions).

d) Define lower level processes needed to operate on the processed input data to produce the output data specified for the upper level processes. Sometimes starting with the processes that produce the output flows and then work your way towards the needed input processes could be easier.

3. The composition of a new data/control flow or store must be known before adding it to the DFD and should be precisely named and recorded in the data dictionary.

4. The total number of flows in a DFD diagram and the number of flows associated with a particular process should be minimized. If a process has too many flows it should be partitioned further into several processes.

The correctness and completeness of the context diagram is an important initial step which must be thoroughly checked and reviewed. All data and control flows represented as external interfaces should be defined in the data dictionary. In the case of an evolutionary development process model, the requirements on the external interfaces might be partially defined. In this case, the interfaces should be specified as much as possible and clear comments should to be attached to those interfaces which are not yet completely specified.

In developing the context diagram emphasis must be put on the simplicity and readability of the diagram. Therefore, abstraction and aggregation are two important techniques that should be utilized to obtain a simpler diagram. External entities and data/control flow can be abstracted by defining supertypes or general types. For example, using general entities such as sensors, actuators, communication equipment, displays, switches etc. help in abstracting external entities and simplifying the diagram. Aggregation is the technique of combining several different entities or data/control flows together in one compound entity or flow. For example, placing a control panel which consists of switches, displays, and lights as one external entity can be one possible way of simplifying the context diagram via aggregation.

The following level of the hierarchy in the Figure is a data flow diagram (entitled DFD0) which represents the major functions outlined in the functional requirements. These functions represent a top level decomposition of the software under developed. Consequently the whole DFD0 is viewed as the child of the process or the bubble representing the system in the top level (i.e., process 0 in the context diagram). Each bubble in DFD0 (numbered 1,2,3, etc.) will have either its own child DFD or a P-spec sheet describing it in more detail as shown in the Figure. DFD1, the child DFD of process 1 in DFD0, has its processes numbered 1.1, 1.2, 1.3, etc. DFDi.j.k.l, i=j=k=l=1,2,3,..., specifies the details of function l in DFDi.j.k in level 3. DFDi.j.k specifies function k in DFDi.j in level 2. DFDi.j specifies function j in DFDi in level 1 which in turn specifies function i in DFD0. Also C-spec sheets are used to specify the controllers shown in each DFD. At any given level i the C-spec sheets for this level should be labeled as i-s1 for controller s1, i-s2 for controller s2, etc., must be specified using a PAT, DT, STD, or an SEM as described above.

Guidelines for Developing the Analysis Model

In developing DFD0 and the lower level DFDs, P-specs, and C-specs, several techniques and guidelines have been suggested in [Shumate&Keller 92] to guide the analyst through the model development process. The objective again as mentioned above to develop a well structured and model with emphasis on simplicity and readability of the specification diagrams. These techniques and guidelines are summarized below:

1. Aggregation. Define a process in a DFD such that it groups together a number of functions from the functional requirements that satisfies the following conditions:
   a) functions that work together to accomplish a specific task. Examples are: the task of interacting with the operator (or processing the pilot requests), the task of controlling the speed of a vehicle, the task of monitoring a set of sensors, the task of generating an alarm, that task of data recording, etc. This condition for defining a process is also termed as the cohesion

accelerate_cmds = [start_accelerate| stop_accelerate]

In the above case the cruise commands are defined to activate, deactivate or resume the operation of maintaining the speed of a vehicle. Start_accelerate and stop_accelerate commands are used for setting the speed to a higher limit.

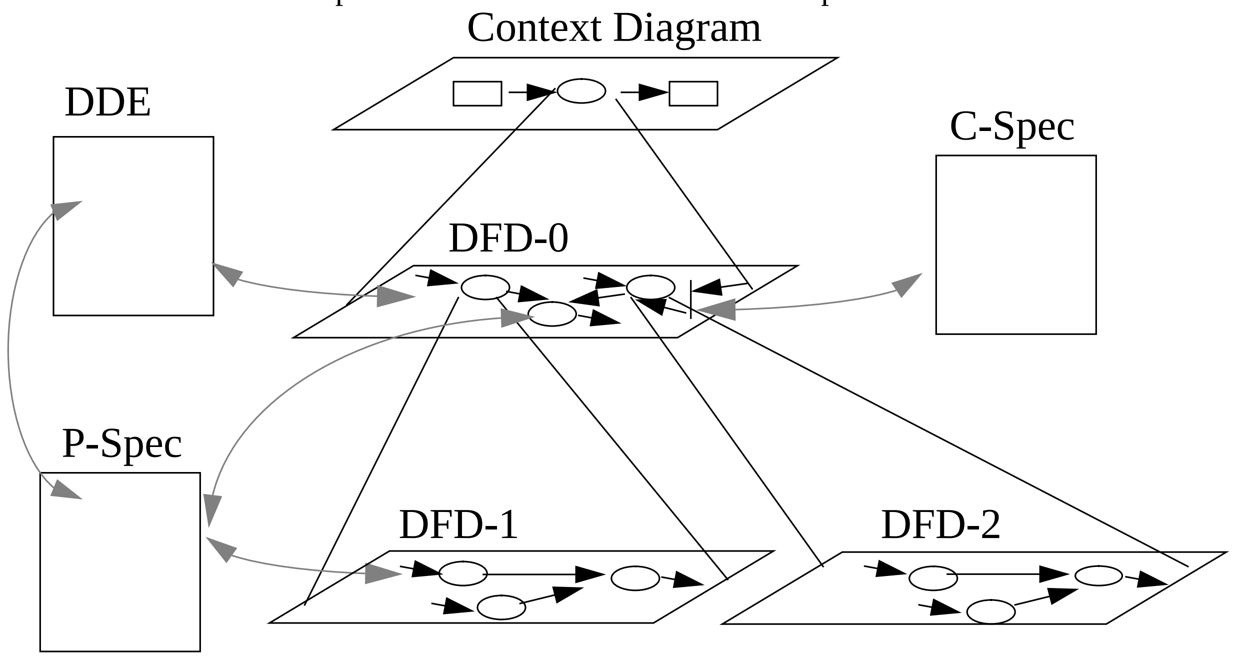
Primitive Information Items

Primitive information items are either continuous or discrete. A continuous information item is characterized by its range, resolution, and units, whereas discrete information items are defined by the set of values or symbols it assumes. The sensor_type data item described in example (1) above is an example of a discrete data item. The sensor_value data item in the same example is a continuous data item whose range and resolution is determined by the sensor type, for example for the temperature sensor the range for the sensor_value can be -20 to 100 degrees F, with a 0.1 degree resolution.

3.2.1.2 The Structured Analysis Model

This section describes the structured analysis models (also called the requirements models) for real-time software systems. Figure 3.9 shows a three dimensional view of the model using the notation described in the previous section. The top layer consists of what is commonly termed as the context diagram (CD). The context diagram is the highest level DFD/CFD in the model. It describes the boundary of the software under analysis as well as the external interfaces and the external entities. The software under consideration is shown as a single process (bubble) in the middle surrounded by input/out data and control flow (representing the external interfaces) from/to the external entities represented by rectangles (termed as terminators) as discussed above.

FIGURE 3.9 Components of SART and their relationship with each other



```
1) sensor_data = sensor_id + sensor_value
   sensor_id = sensor_type + sensor_No
   sensor_type = [“Temp” | “Pressure” | “Fuel” |  “Smoke”]
                 * four types of sensors are assumed, these are temperature sensors,
                  engine pressure sensors, fuel capacity sensors, and smoke detection
                  sensors *

   sensor_No = 2{decimal_digit}5
                 *the sensor number is 2 to 5 decimal digits*
```

The above example defines sensor_data, sensor_id, and sensor_No as compound data items. Primitive data items are sensor_value, sensor_type (which has one of four possible symbols), and decimal_digit (which can be defined as a data item with one of ten possible symbol 0-9)

```
2) objects = [coins| slugs]
   *defines objects detected by a vending machine*

   coins = {[quarters|dimes|nickles]}8
   *defines the types and from 0-8 coins can be            expected*

3) alert_queue = {sensor_id}
   * defines a queue consisting an unlimited number of objects specified by the
     sensor_id data defined in example 1) above*

4) cruise_cmds = [[activate|resume]
                 + ([start_accelerate | stop_accelerate])
                 | deactivate]
```

*cruise commands are defined as either activate/resume or deactivate. The activate command starts the speed control function where the current speed of the vehicle is maintained, the resume command is used after cruising was deactivated to maintain the previously selected speed. If either activate or resume are selected they can be followed by an optional command which is either start_accelerate or stop_accelerate.

Complex expressions can be obtained when the selection, concatenation, and optional operations are combined as shown in the above example. The above example shows that a start_accelerate or a stop_accelerate commands must always be preceded by an activate or a resume command which could be contradicting or violating the system requirements. It should be emphasized that simplicity and readability are important criteria that should be followed to avoid errors when defining compound information items. The information item specified in example 4 above can be redefined by breaking it into two items to allow for simpler expressions as follows:

```
cruise_cmds = basic_cmds + (accelerate_cmds)
 basic_cmds =[activate| resume| deactivate]
```

3.2.1.1.4 The Data Dictionary

The data dictionary contains the definition of all the information items consisting of flows as well as the stores both for data and controls.

Information items, in general, are divided into two types: primitive data items; and compound data items. Primitive information items are those items not composed of any other data items. Compound data items on the other hand may be composed of other compound data items and/or primitive data items. Examples of primitive data items are, a temperature sensor reading, a binary switch reading, operation status, or an identification number. Examples of compound data items are operator command which consists of several different types of commands, sensor data consisting of the readings of different sensors, etc. Compound data are needed for the purpose of abstraction to define the data hierarchy, which makes the analysis diagrams and tables easily manageable and readable. The DFDs functional hierarchy usually requires also some form of data hierarchy where compound information items are used in the higher levels of hierarchy and primitive information items are used in the lower levels.

Compound Information Items

The notation used by many ICASE tools for defining compound information items are described below. The following table presents the symbols used in defining compound information items.

Symbol	Meaning	Description
=	Composed of	used to define a compound information item in terms of other information items.
+	together with (concatenation)	used to specify the set of(concatenation)components of an information item (similar to a concatenation operation).
M{}N	M to N iterations of what is enclosed in {}	used to define repetitive occurrences of components defined inside the curly {}. M and N are used to define a the lower and upper limits on number of iterations, respectively. If any or both are omitted, the default for M is zero, and for N is unlimited number.
[|]	select one of	square brackets containing several components separated by | defines an expression in which exactly one of the specified components are selected at any given instance of the information flow.
( )	Optional	expressions between parentheses may or may not appear in the flow instance.
“ "	Literal	used to define literal symbols (e.g., “TRUE”, “Temp”, “Fuel”, see the examples below).
* *	Comment	text describing the data flow should be enclosed in asterisks.

Table 2:

Examples of compound information items are as follows:

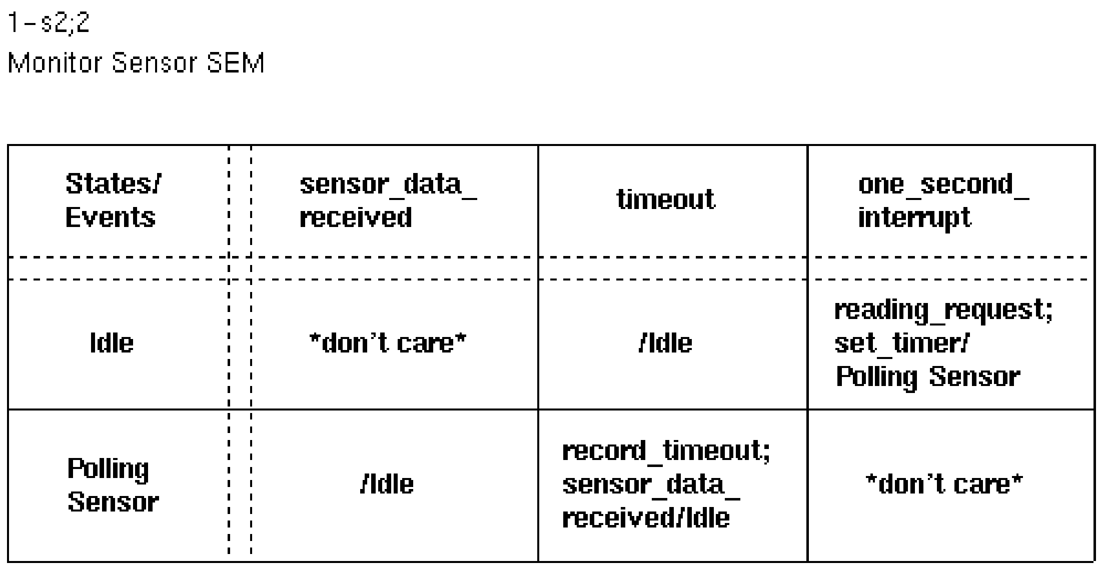
handle for complex controllers, where the number of states and/or the number of transitions are large. SEMs contain the same information contained in STDs but in a tabular form. It can specify a sequential controller with a large number of states.

Each row in an SEM corresponds to a particular state of the controller. The set of columns consists of event columns followed by an actions column. Each input of the controller is represented by an event column in the table. This cells in the table for the actions column specify the actions performed for each state (following the Moore model) numbered by their order of execution.

The cells in the table for the event columns are filled to specify what happens when a given state receives a particular event. A cell contains the set of actions performed (following the Mealy model) numbered by their order of execution followed by a slash and the name or number (i.e., row number) of the next state to be transitioned to the particular state-event combination which the cell represents. This corresponds exactly to a labeled transition in the STD.

Note that the SEM contains cells in the events columns for all state-event combinations. In cases in which the event at a particular state is “ignored” or simply “can't happen” the cell is left blank with no specifications of actions or next state. This is the main difference between SEMs and STDs. STDs does not show the event ignored or cant' happen cases at all since it only specify transitions representing recognized and serviced events. This is why SEMs are more suitable for verification than STDs since the analyst must give a consideration to each state-event combination. “Event ignored” cases simply mean that the event is consumed and nothing happens, whereas “Can't happen” specify the case in which the state-event combination can never occur in reality. Since this might be a source of errors in the analysis, the analyst should provide a note explaining why (unless it is very obvious) such combination can never happen or such an event should be ignored.

FIGURE 3.8 An example of an SEM

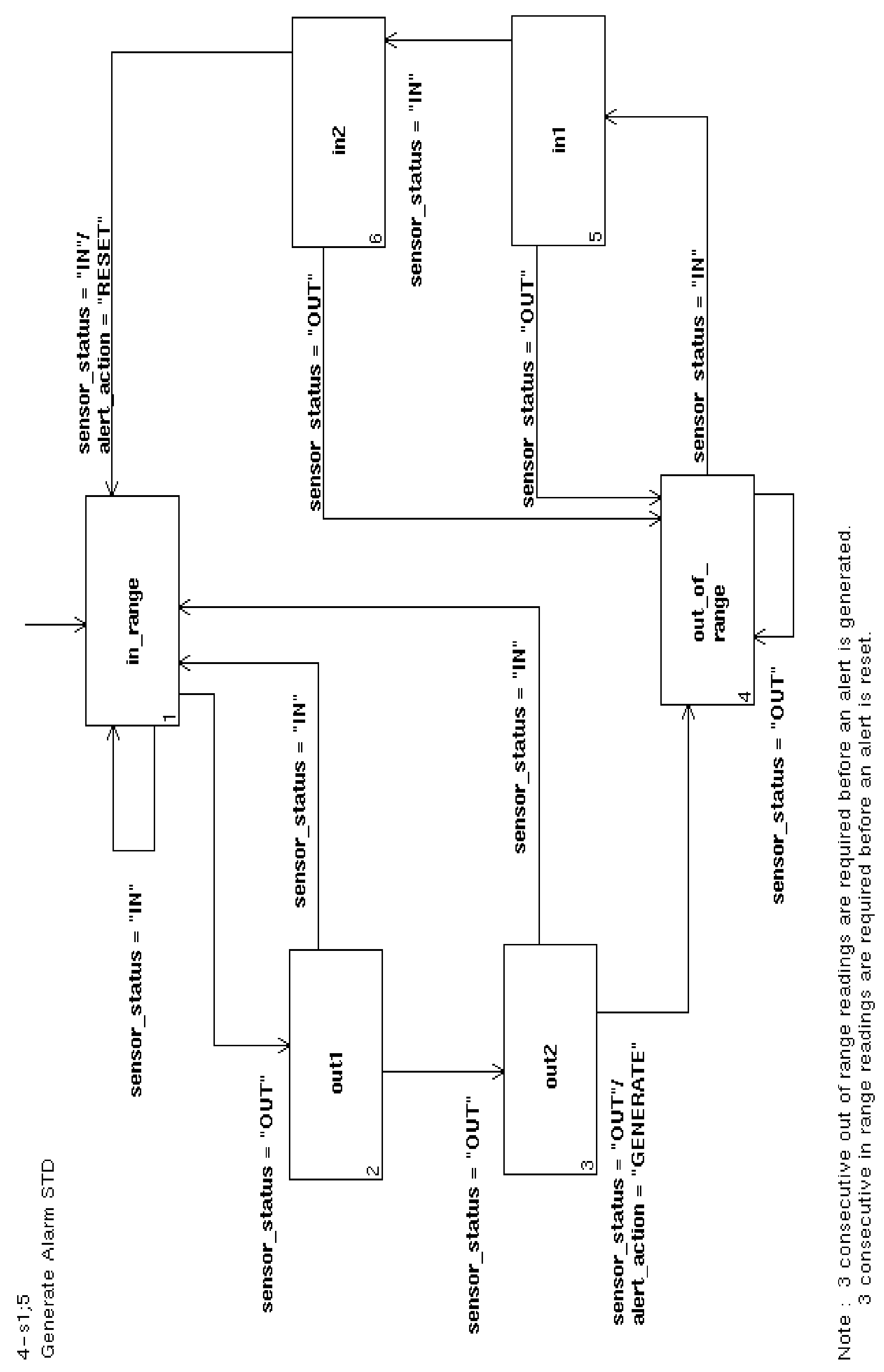


1-s2;2

Monitor Sensor SEM

States/ Events	sensor_data_ received	timeout	one_second_ interrupt
Idle	*don't care*	/Idle	reading_request; set_timer/ Polling Sensor
Polling Sensor	/Idle	record_timeout; sensor_data_ received/Idle	*don't care*

FIGURE 3.7 Example of an STD



State/Event Matrices (SEMs)

STDs are easy to draw and understand, in fact they are referred to as the most expressive representation of sequential controllers, however, they quickly become very difficult to

Every STD contains an initial state in which the control processes starts when the controller is activated. This state is designated by a transition specified by an incoming directed arc with no source or origin (i.e., incoming or originating from the empty space).

An important note, concerning process activation signals specified as actions in Mealy STDs, is that activation signals associated with a transition are assumed to continue in effect until the next transition occurs. This means that an activated process will remain active until the next transition occurs. There is no need for deactivation signals since processes are automatically deactivated, unless they are activated again in the next transition. For Moore STDs, a process activated at a given state will remain active until a transition to different state in which the process in not activated occurs.

FIGURE 3.6 An example of a PAT.

1-s1;4
Monitor Sensor PAT

sensor_data_ received	fuel_data_ received	record_ timeout	2	3	1
"TRUE"			1		
	"TRUE"			1	
		"TRUE"	2		1

State Transition Diagrams (STDs)

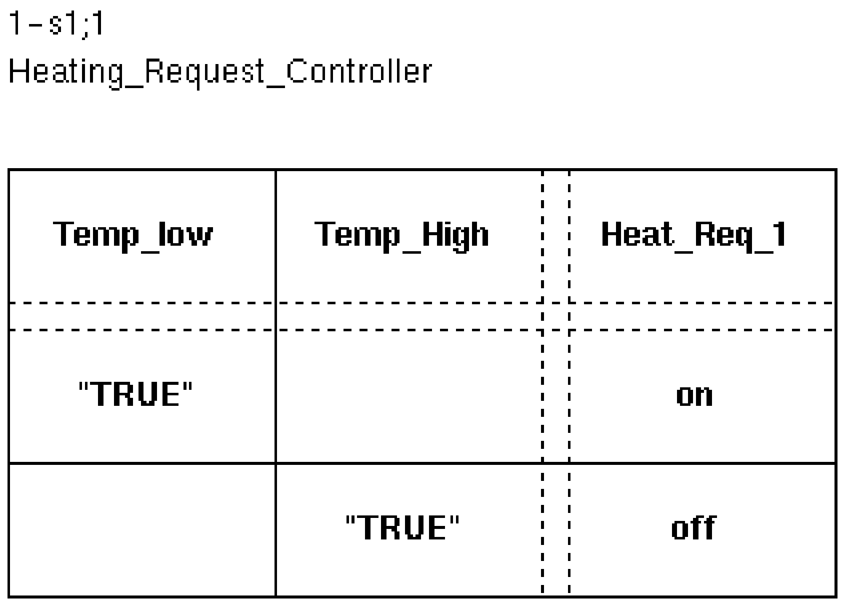
STDs specify controllers consisting of a sequence of states for sequential controls. A rectangle is used to define each state and directed arcs between rectangles specify transitions from one state to another. A state transition is caused by a specific event consisting of a combination of input control values and produces actions consisting of process activations and a combination of output control values. A PAT can be used with a STD to specify the process activations for the various state transitions.

There are two well known sequential machine models used to define sequential controllers. These are the Moore model and the Mealy model. In a Moore type model, the output actions (process activations and output control value) are associated with the states, whereas in a Mealy model the actions are associated with the transitions between states (i.e., with state-event combinations).

STDs can be of the Mealy type where the output controls and process activations are associated with the state transitions. In this case a transition is labeled by the event causing it followed by a slash followed by a set of numbered actions. The number given to an action specify the order in which it is executed (e.g, a value is specified for an out put control flow, or an activation signal is asserted). STDs can also be of the Moore type (in which the output controls and process activations are associated with the states themselves), in which case a state definition rectangle consists of a state name followed a slash followed by the set of numbered actions. It is possible to mix the two notations in one STD, in which case a hybrid Mealy/Moore model is defined. The model will be developed depending on whether it is more convenient to specify actions (i.e, generating outputs or activating processes) within the rectangles defining states or specify actions with the arcs representing transitions. Each output action must be specified with a state (or a set of states) or with a transition.

use of don't cares (blank entries) simplifies the table. The control outputs specified may contain process activation signals.

FIGURE 3.5 An Example a DTS



1-s1;1

Heating_Request_Controller

Temp_low	Temp_High	Heat_Req_1
"TRUE"		on
	"TRUE"	off

Process Activation Tables (PATs)

This type of C-specs is also used to specify a combinational controller which has no explicit output controls. It used to specify process activation for a given combination of input controls. A PAT is a special case of a DT in which the names of the processes to be activated are specified instead of the output control flows.

FIGURE 3.4 P-Spec of Process 1.8

```
NAME:   1
TITLE:   Check Security Code


 INPUT/OUTPUT:
 Customer_Card_Info: data_in
 card_identification_number: data_in
 security_code: data_in
 customer_id: data_out
 message: data_out

 BODY:
 Look up the card_identification_number in the Customer_Card_Info
 store. If the atm card's security code matches the <user_entered>
 security_code, send out the corresponding customer_id. If the
 <user_entered>security_code does not match the assigned security_
 code, send an <error>message to the Customer.
```

3.2.1.1.3 Control Specifications (C-specs)

A control node (or a controller) in the CFD is specified by a control specification artifact called C-spec. A C-spec determines in detail how when the out control flows of the control node are asserted. It also specifies the currently activated processing nodes in the corresponding DFD. The activation signals to the processing nodes are not shown explicitly as outputs of the control node.

The notations for C-specs are divided into four different types:

- Decision Tables (DTs);
- Process Activation Tables (PATs);
- State Transition Diagrams (STDs);
- State/Event Matrices (SEMs).

The choice of a particular type depends on the control node of the problem at hand. DTs are used to specify combinational controllers which assumes only one state. A STD or a SEM are used for specifying sequential controllers consisting of several states. PATs are used to specify combinational control nodes used for activation of processing nodes. A controller specified by a PAT should have no output control flows (activation signals are not considered to be explicit output flows of control nodes). These types are further described in the following discussion.

Decision Tables (DT)

A DT is similar to a truth table in combinational digital circuit design except that here inputs and outputs do not have to be logic variables but can be any discrete variable defined over a finite set of values. DTs specify combinational controllers (i.e., controllers with only one state). Each row in a DT specifies the values for the output control items for a combination of input control items. The combination of inputs not specified by a row in the table are assumed to be don’t cares. Also specific inputs or outputs which are left blank in any given raw are also considered don't cares. Don't care conditions arise from the combination of inputs that can never occur or from the same output produced by several combinations of inputs (in which case some the inputs are specified as don't cares). The

- A processing node can never consume a control flow item except when it is defined by a lower level DFD/CFD containing a control node which actually consumes the control flow item.
- A processing node can produce a control flow item using input data flows. For example a processing node which monitors temperature readings (temperature reading is an input data flow) can produce a control flow with a true or false value depending on the temperature value being greater than a given threshold temperature.

3.2.1.1.2 Process Specification (P-specs)

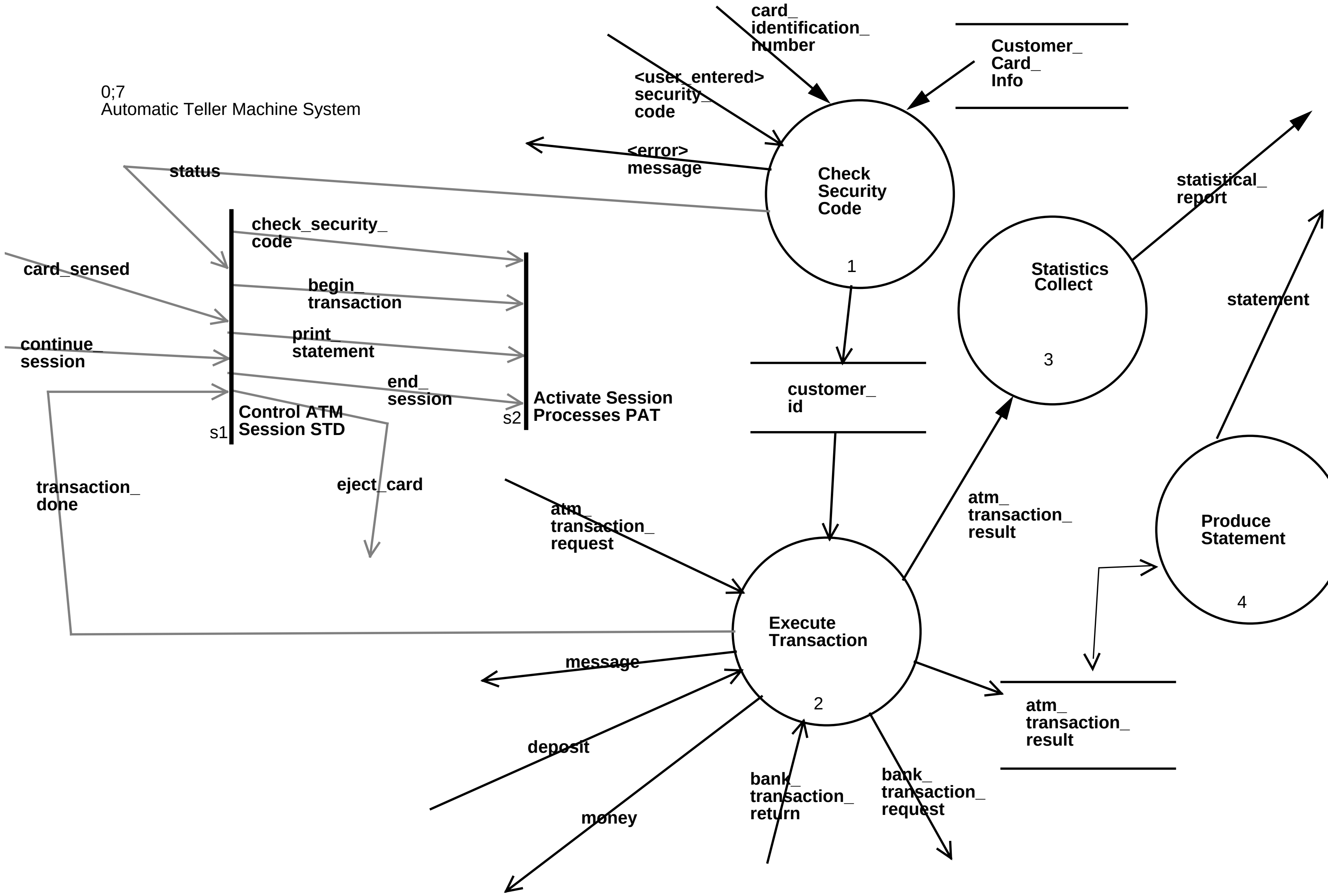
Each primitive processing node in a DFD diagram (i.e., a simple processing node which does NOT have a child DFD in a lower level to specify it in finer detail) must have a P-spec. A P-specs in essence is the child of the primitive processing node it describes. It shares the same input and output flows and specifies how the produced output flows are obtained from the consumed input flows.

P-Specs are defined in terms of textual specification describing a process that is procedural. Structured English is used to make the text concise. The intent of structured english is to combine the rigor of a programming language and the readability of English. The following four structures can be used in a P-spec:

- Concurrency: Statements under this section specify operations on input data or computations of output data which can be done in parallel.
- Sequence: this section or structure specifies sequential operations. The statements in the section after the very first statement should start with the clause “next” or the clause “then” to emphasize the sequential relation between operations.
- Decision: This section consist of “if-then-else” statements to specify conditional operations.
- Repetition: this section consists of “while” a “for”, a “repeat-until”, or a “cycle” clause to specify a repeated section of operations.

P-specs can also contain mathematical equations and illustrations such as tables, diagrams, and graphs to ensure that specifications are complete and unambiguous.Figure Figure 3.4 shows a P-spec for processing node 1 in Figure 3.2.

FIGURE 3.3 DFD/CFD of an ATM Network



- A control flow information item, on the other hand, is consumed by a control node to be used in activating and deactivating processing nodes, or in producing out going control flows.

-
-
- **A data store** (two solid horizontal bars), represents an information repository ranging from a simple data buffer or queue to a large data base.
-
- **A control store** (two dashed horizontal bars) represents a memory component containing information items used by controllers (as control flows).

Figure 3.3 shows an example of a DFD/CFD. The diagram specifies the data flow, control flow, and the major data processing and control activities in an Automatic Teller Machine (ATM). Four data processing nodes (bubbles), two control nodes, and three data stores are shown. The first node checks the customer card number and security code against the customer information database. The node produces a control flow signal called status which specifies whether the card and the code were approved or not. If not approved, an error message is sent to the customer. If approved, the customer ID information is stored in a data store to be used by other nodes.

The second node in Figure 3.3 executes transactions requested by the customer. This node produces statistics sent as a data flow to node 3 and stores the current transaction result in a data store to be used by node 4 to print the statement for the customer at the end of the ATM session. Control node S1 senses the card_sensed control flow input and if true it produces a true on the output flow labeled check_security_code. This flow is used by control node S2 to activate processing node 1. S1 then waits for the status control signal produced by processing node 1. If the value of this signal specifies that the card is approved, S1 produces a true on the begin_transaction output flow. S2 will use this flow to activate processing node 2. When processing node 2 is done with the current transaction it generates a true on the transaction_done output control flow. This flow is then used by S1 to sense the continue_session input flow. If it is true, it produces a true on the begin_transaction output flow used to start another transaction by S2. If continue_session is false, S1 produces a true on both end_session and print_statement output flows used by S2 to activate processing node 4 and terminate the ATM session.

Control Flows vs. Data Flows

One of the major confusing items in DFDs/CFDs is the difference between control flow and data flow information items. By definition, any information item used directly for controlling the data processing activities or is specified as an input or an output of a control node must be designated as a control flow information item, otherwise the information item is a data flow. The following remarks should help in clearing up the confusion,

- A data flow information item is consumed by a processing node to be used for computing the value of an expression.

The notation for DFDs/CFDs consists of the following components as shown in Figure 3.2:

- **External Entities** (represented by rectangles, also called terminators to terms), specify entities which are outside the boundaries of the system (or the CSCI) to be modeled. These entities are either hardware components or software components or other systems which produce (or consume) information needed (or produced) by the modeled system. Examples of external entities are operators, sensors, actuators, etc.
-
- **processing nodes** (represented by circles or a bubbles), specify processing functions within the boundaries of the modeled system. The naming of nodes should represent the actions performed and should always start with a verb followed by an object on which the verb acts. For example, names such as poll sensors, input pilot commands, display pilot data, etc. are all correct names for processing nodes. A processing node operates on a set of input data or control information and produces a set of output information which might also be data or control information. A processing node represents a transformation at a given level of abstraction. If the transformation is complex, the processing node is associated with (i.e., is a parent of) a lower level DFD/CFD (also called the child diagram) which represents the transformation in a finer level of detail. This hierarchical nature of structured analysis model will be explained further in the next section.
-
- **data flows** (represented by solid lines) are directed links originating (or produced) from an external entity, a processing node, or a data store and terminating (or consumed by) at an external entity, a processing node, or a data store. (see the a, b, c cases in the Figure). A data flow may represent a single data item or a group of data items abstracted or combined under one name. A group data flow associated with an input or an output of a parent processing node may be divided into its element data items at the child DFD/CFD. Data flows are described in details in terms of its elements and the values they might take in the data dictionary of the structured analysis model.
-
- **control nodes** (represented by vertical bars) represent controllers which activate processes specified by processing nodes in the same diagram, and consume and generate control signals (or control flows). A control node symbolizes a part of the control section in any computing system which takes care of controlling the data processing elements in the processor. The details of the control process in a control node are specified in another sheet by a state transition diagram and/or a table (e.g., a process activation table, decision table, or a state-event table). Every control node is labeled to distinguish it from other control nodes in the same control flow diagram.
-
- **control flows** (represented by dashed lines) is a directed link originating from an external entity, a processing node, or a control node, and terminating at a control node, or an external entity (a control flow is also allowed to terminate at a processing node only if this node is defined by a lower level DFD/CFD having a control node which consumes the control flow).

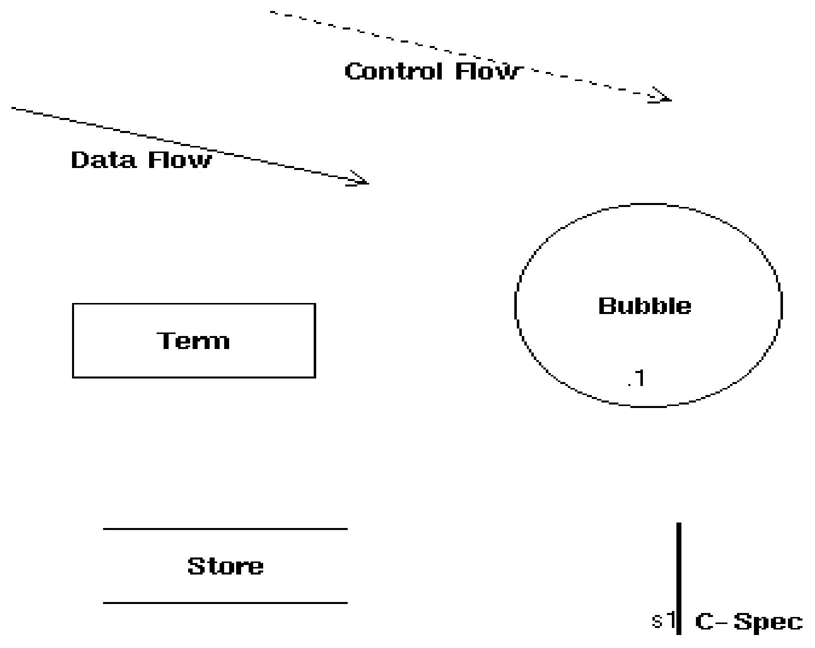
- Entity-Relationship Diagrams (ERDs) provide an information model for the data items and control signals and the relationships among these data items. It used to facilitate the representation and specification of information items.

Information modeling and ERDs will be explained in detail in section 3.3 on Object-Oriented Analysis (OOA). The first four artifacts above will be described in the following subsections.

3.2.1.1.1 Data Flow and Control Flow Diagrams (DFDs/CFDs)

Data flow diagrams (DFDs) specify the data processing activities in a system. A control flow diagram (CFD) specifies the flow and the processing of control information in the system. DFDs consist of processing nodes and data flows and data stores, whereas CFDs consist of control nodes, control flows and control stores. DFDs and CFDs are tightly coupled since the control nodes in a CFD controls the activation of processing nodes in its corresponding DFD, which in turn might produce a control flow used as an input for a control node in the CFD. The two diagrams can be drawn separately for very complex systems. However in hierarchical analysis the emphasis is to keep the combined DFD/CFD diagram simple at any given level of the hierarchy and divide the complexity among several lower level DFD/CFD diagrams. In the sequel we will explain the notation of the combined DFD/CFD diagram.

FIGURE 3.2 Objects used in Structured Analysis



requirements or refinements of incomplete requirements in an evolutionary development process.

- Problems of size must be dealt with using the effective method of partitioning. This is precisely why hierarchical functional decomposition is one of the crucial steps in this method.
- Graphics must be used whenever possible. This is exemplified by the maze of diagrams needed to specify the functional levels in large problems.
- We have to differentiate between the logical (essential) and physical (implementation) considerations. Engineers are often tempted to include implementation details early to further specify the requirements and narrow down the design alternatives.

Demarco then proceeded further to establish requirements for the structured analysis method as follows: the method should help us partition our requirements and document that partitioning before specification; it should give us means of keeping track of and evaluating interfaces; it should facilitate the development of new tools to describe logic and policy better than narrative text.

3.2.1.1 The ICASE Notation for Structured Analysis

The notation used for creating structured analysis diagrams by most ICASE tools are introduced in this section. This notation is based on the work of Hatley and Pirbhai [HAT 87] mentioned briefly in the section 3.1.2. As mentioned in this section, the notation consists of following types of artifacts:

- Data Flow and Control Flow Diagrams (DFDs/CFDs). These diagrams model the processing of information in terms of data and control flows; data processing nodes and control nodes (or controllers) are defined to represent the data processing functions and the control functions, respectively, from the requirements of the system under analysis. Data flow information items are those information items received and processed by processing nodes, whereas control flow information items are used by the control nodes to control the data processing activities in the system at hand.
- Process specifications are used to described the details of the data processing nodes defined in a DFD. These specifications consist of scripts of pseudo-code or just plain text which explains how the output flows of a particular processing node is generated from its input flows;
- Control specifications are used to describe the details of controls nodes (or controllers) in a CFD. These specifications define the behavioral (or state) model of the controllers and specify how the output control flows are obtained from the input control flows. They also specify when the data processing nodes are activated or deactivated.
- Data Dictionary which defines all the information flows and the data and control stores in the system. It contains text hat defines each information item and its value range.

3.2 Structured Analysis for Real-Time Software Using ICASE

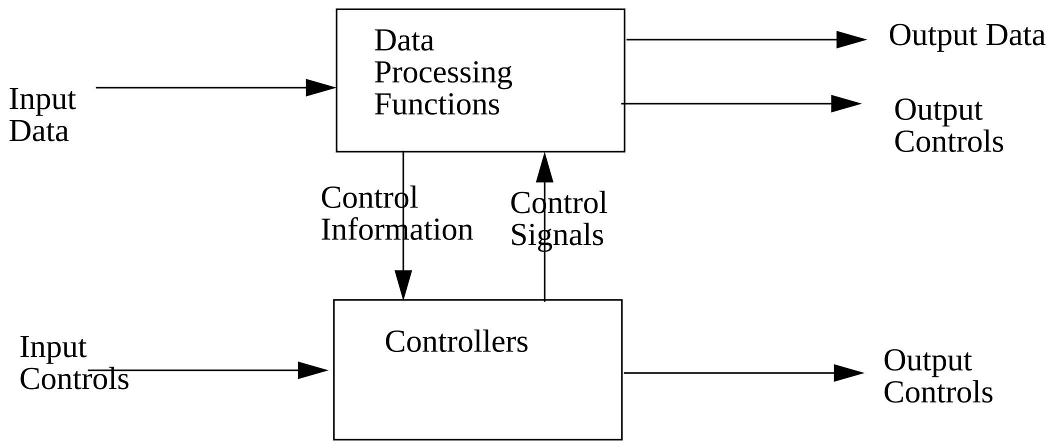
The structured analysis technique introduced briefly in section 3.1.1 is discussed in detail in this section. ICASE tool support given in Teamwork/RT will be illustrated using examples. The section starts by introducing the notation used, the supported features in Teamwork/RT are described, and several examples are presented at the end of the section. These examples are obtained from the Samples directory of Teamwork.

3.2.1 Introduction to Structured Analysis

In his well known book on Software Engineering, Roger Pressman [PRESS 93, pp207-208] describes structured analysis as follows:

“Structured analysis, like all software requirements analysis methods, is a model building activity. Using a notation that is unique to the structured analysis method, we create models that depict information (data and control) flow and content, we partition the system functionally and behaviorally, and depict the essence of what must be built. Structured analysis is not a single method applied consistently by all who use it. Rather it is an amalgam that has evolved over almost 20 years.”

FIGURE 3.1 Structured Analysis Methodology



There are probably no other software engineering method that has generated as much interest, been tried (and often rejected and tried again) by many people, provoked as much criticism, and sparked as much controversy. But the method has prospered and has gained widespread use throughout the software engineering community.

Tom Demarco, one of the pioneers of structured analysis, described the need for such a method more than 16 years ago [DEM 79, pp15-16]. He suggested to make the following additions to the analysis phase goals to overcome recognized problems and failings of the analysis phase:

- The products of analysis must be maintainable. This applies particularly to the target document (i.e., the Software Requirements Specification document). Maintainability here is very much needed to facilitate changes due to changing